Module -I

Introduction to Compiling:

1.1 INTRODUCTION OF LANGUAGE PROCESSING SYSTEM

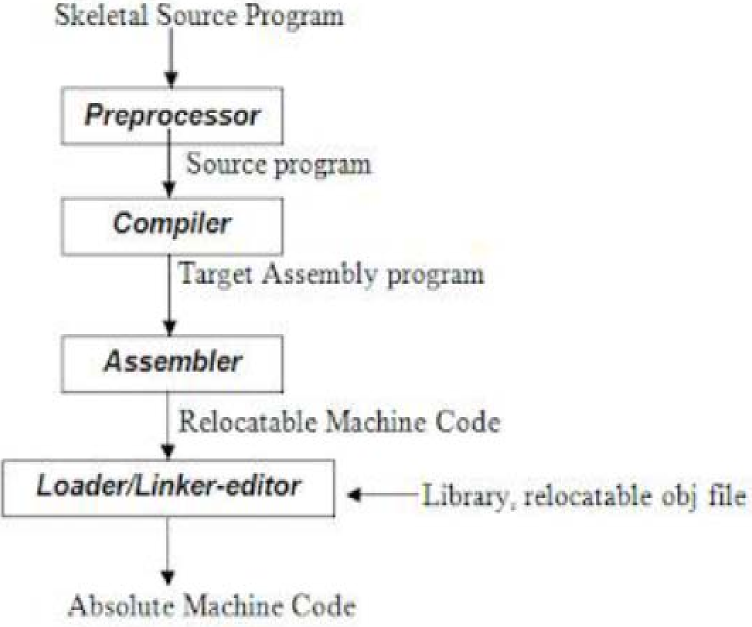


Fig 1.1: Language Processing System

Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

COMPILER

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.

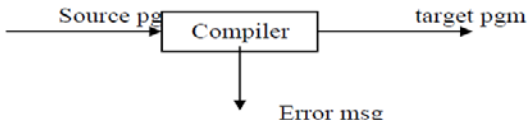


Fig 1.2: Structure of Compiler

Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.

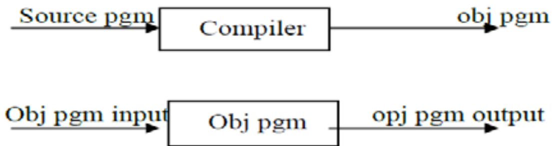


Fig 1.3: Execution process of source program in Compiler

ASSEMBLER

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER

An interpreter is a program that appears to execute a source program as if it were machine language.

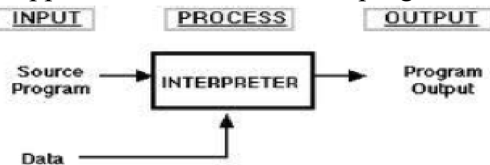


Fig1.4: Execution in Interpreter

Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.

Type of object that denotes a various may change dynamically.

Debugging a program and finding errors is simplified task for a program used for interpretation.

The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is *slower*.

Memory consumption is more.

LOADER AND LINK-EDITOR:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it,

thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could"relocate" directly behind the user's program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

1.2 TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the HLL program input into an equivalent ml program.

2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

1.3 LIST OF COMPILERS

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

1.4 STRUCTURE OF THE COMPILER DESIGN

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called **'phases'**.

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

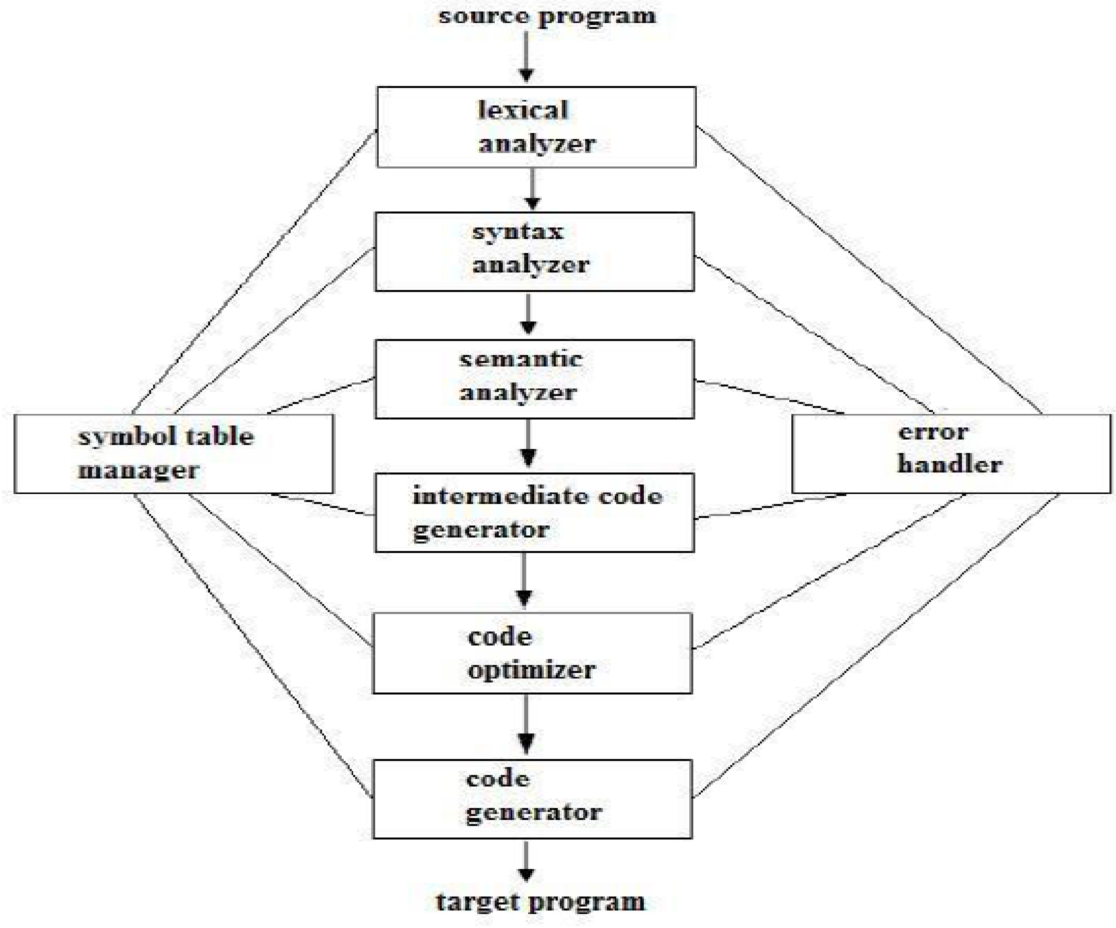


Fig 1.5: Phases of Compiler

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:- This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a 'Symbol Table'.

Error Handlers:-
It is invoked when a flaw error in the source program is detected. The output of LA is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language. It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression. Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)
1, divide A by B and then multiply by C or
2, multiply B by C and then use the result to divide A.
each of these two interpretations can be represented in terms of a parse tree.

Intermediate Code Generation:-
The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands. The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization
This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

a. Local Optimization:-

There are local transformations that can be applied to a program to make an improvement. For example,

If **A** > **B** goto **L2**

Goto **L3**
L2 :

This can be replaced by a single statement
If **A < B** goto **L3**

Another important local optimization is the elimination of common sub-expressions
A := B + C + D
E := B + C + F
Might be evaluated as

T1 := B + C
A := T1 + D
E := T1 + F
Take this advantage of the common sub-expressions **B + C.**

b. Loop Optimization:-
Another important source of optimization concerns about **increasing the speed of loops**. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

Code generator :-
Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book-keeping :-
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Error Handing :-
One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:

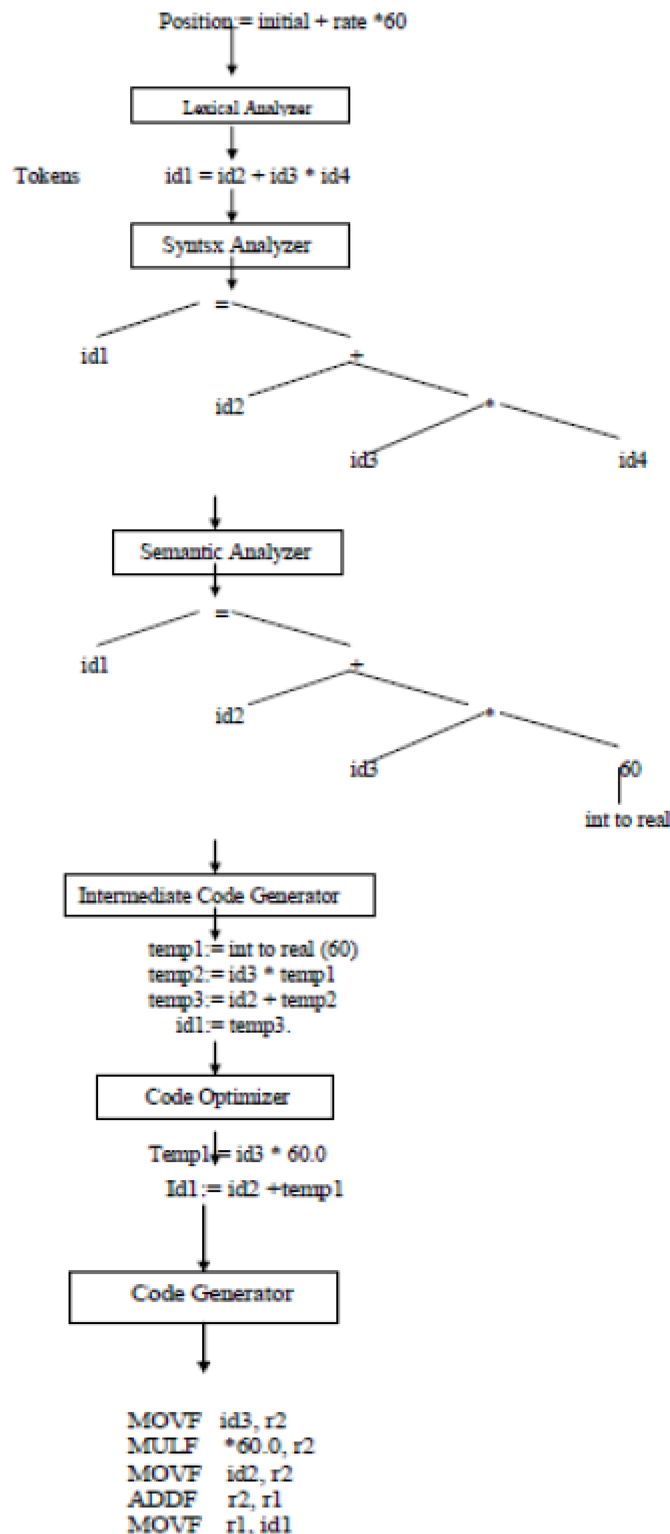


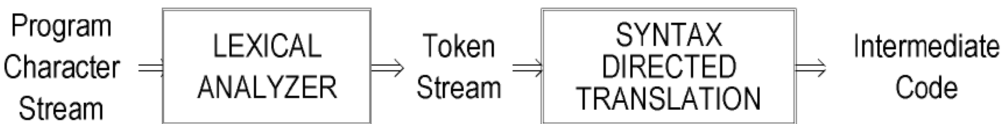
Fig 1.6: Compilation Process of a source code through phases

2. A simple One Pass Compiler:

2.0 INTRODUCTION: In computer programming, a **one-pass compiler** is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

2.1 OVERVIEW

- Language Definition
  - Appearance of programming language :
    Vocabulary : Regular expression
    Syntax : Backus-Naur Form(BNF) or Context Free Form(CFG)
  - Semantics : Informal language or some examples



- **Fig 2.1.** Structure of our compiler front end

2.2 SYNTAX DEFINITION

- To specify the syntax of a language : CFG and BNF
  - Example : if-else statement in C has the form of statement → if (expression) statement else statement
- An alphabet of a language is a set of symbols.
  - Examples : {0,1} for a binary number system(language)={0,1,100,101,...}
    {a,b,c} for language={a,b,c, ac,abcc..}
    {if,(,),else ...} for a if statements={if(a==1)goto10, if--}
- A string over an alphabet
  - is a sequence of zero or more symbols from the alphabet.
  - Examples : 0,1,10,00,11,111,0202 ... strings for a alphabet {0,1}
  - Null string is a string which does not have any symbol of alphabet.
- Language
  - Is a subset of all the strings over a given alphabet.
  - Alphabets Ai Languages Li for Ai

Alphabets Ai	Languages Li for Ai
A0={0,1}	L0={0,1,100,101,...}
A1={a,b,c}	L1={a,b,c, ac, abcc..}
A2={all of C tokens}	L2= {all sentences of C program }

- Example 2.1. Grammar for expressions consisting of digits and plus and minus signs.
  - Language of expressions L={9-5+2, 3-1, ...}
  - The productions of grammar for this language L are:

list → list + digit
list → list - digit
list → digit
digit → 0|1|2|3|4|5|6|7|8|9

- list, digit : Grammar variables, Grammar symbols
- **0,1,2,3,4,5,6,7,8,9,-,+** : Tokens, Terminal symbols

- Convention specifying grammar
  - Terminal symbols : bold face string **if, num, id**
  - Nonterminal symbol, grammar symbol : italicized names, list, digit ,A,B

- Grammar G=(N,T,P,S)
  - N : a set of nonterminal symbols
  - T : a set of terminal symbols, tokens
  - P : a set of production rules
  - S : a start symbol, S∈N
  -
- Grammar G for a language L={9-5+2, 3-1, ...}
  - G=(N,T,P,S)
    N={list,digit}
    T={0,1,2,3,4,5,6,7,8,9,-,+}
    P : *list -> list + digit*
    list -> list - digit
    list -> digit
    digit -> **0|1|2|3|4|5|6|7|8|9**
    S=*list*

- Some definitions for a language L and its grammar G
  - Derivation :
    A sequence of replacements $S\Rightarrow\alpha 1\Rightarrow\alpha 2\Rightarrow\ldots\Rightarrow\alpha n$ is a derivation of αn.
    Example, A derivation 1+9 from the grammar G
    - left most derivation
      list ⇒ list + digit ⇒ digit + digit ⇒ 1 + digit ⇒ 1 + 9
    - right most derivation
      list ⇒ list + digit ⇒ list + 9 ⇒ digit + 9 ⇒ 1 + 9
  - Language of grammar L(G)
    L(G) is a set of sentences that can be generated from the grammar G.
    $L(G)=\{x| S \Rightarrow^* x\}$ where x ∈ a sequence of terminal symbols
  - Example: Consider a grammar G=(N,T,P,S):
    N={S} T={a,b}
    S=S P ={S → aSb | ε }
    - is aabb a sentecne of L(g)? (derivation of string aabb)
      S⇒aSb⇒aaSbb⇒aaεbb⇒aabb(or S⇒* aabb) so, aabbεL(G)
    - there is no derivation for aa, so aa∉L(G)
    - note L(G)={anbn| n≧0} where anbn meas n a's followed by n b's.

- **Parse Tree**

A derivation can be conveniently represented by a derivation tree(parse tree).

- The root is labeled by the start symbol.
- Each leaf is labeled by a token or ε.
- Each interior none is labeled by a nonterminal symbol.
- When a production A→x1… xn is derived, nodes labeled by x1… xn are made as children
  nodes of node labeled by A.
    - root : the start symbol
    - internal nodes : nonterminal
    - leaf nodes : terminal

- Example G:
  list -> list + digit | list - digit | digit
  digit -> 0|1|2|3|4|5|6|7|8|9
    - left most derivation for 9-5+2,
      list ⇒ list+digit ⇒list-digit+digit ⇒ digit-digit+digit ⇒ 9-digit+digit
      ⇒ 9-5+digit ⇒ 9-5+2
    - right most derivation for 9-5+2,
      list ⇒ list+digit ⇒list+2 ⇒list-digit+2 ⇒ list-5+2
      ⇒ digit-5+2 ⇒ 9-5+2

  parse tree for 9-5+2

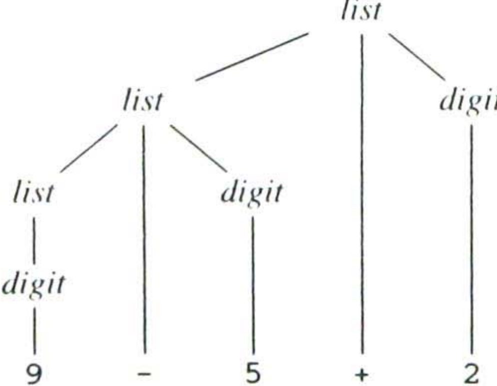


Fig 2.2. Parse tree for 9-5+2 according to the grammar in Example

Ambiguity

- A grammar is said to be ambiguous if the grammar has more than one parse tree for a given string of tokens.
- Example 2.5. Suppose a grammar G that can not distinguish between lists and digits as in Example 2.1.
    - G : string → string + string | string - string |0|1|2|3|4|5|6|7|8|9

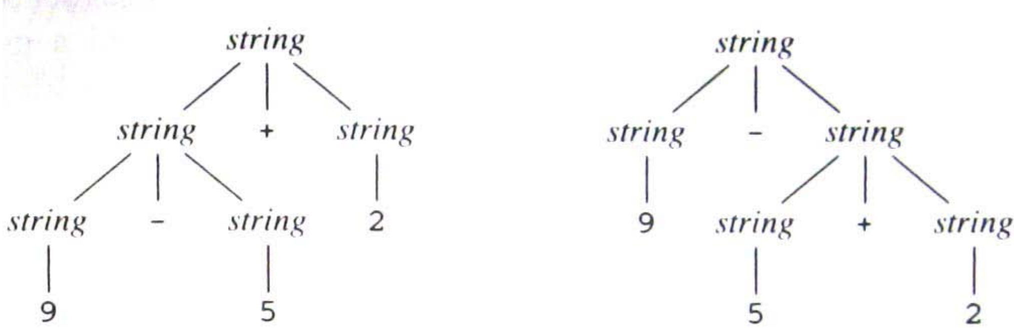


Fig 2.3. Two Parse tree for 9-5+2

- 1-5+2 has 2 parse trees => Grammar G is ambiguous.

Associativity of operator

A operator is said to be left associative if an operand with operators on both sides of it is taken by the operator to its left.

eg) 9+5+2≡(9+5)+2, a=b=c≡a=(b=c)

- Left Associative Grammar :

  list → list + digit | list - digit

  digit →0|1|…|9

- Right Associative Grammar :

  right → letter = right | letter

  letter → a|b|…|z

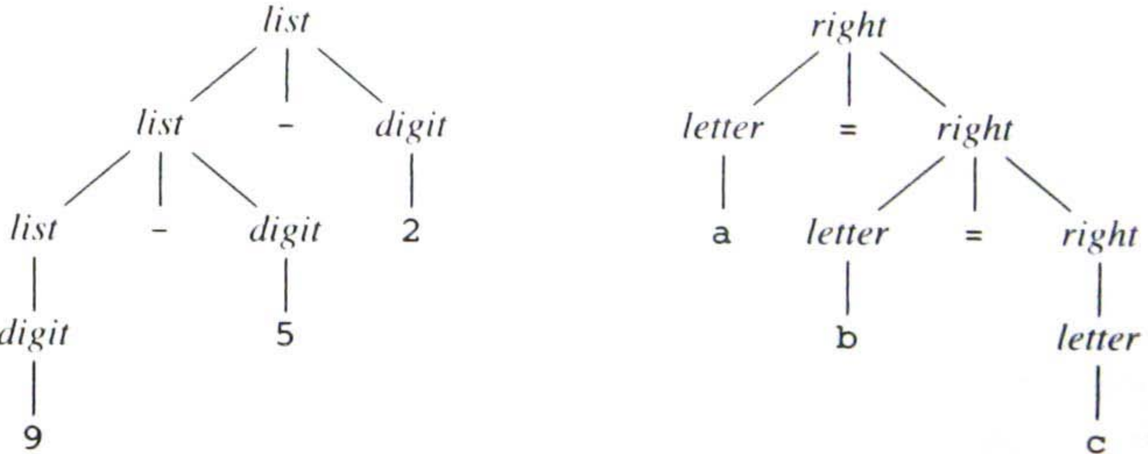


Fig 2.4. Parse tree left- and right-associative operators.

Precedence of operators

We say that a operator(*) has higher precedence than other operator(+) if the operator(*) takes operands before other operator(+) does.

- ex. 9+5*2≡9+(5*2), 9*5+2≡(9*5)+2
- left associative operators : + , - , * , /
- right associative operators : = , **

- Syntax of full expressions

operator	associative	precedence
+ , -	left	1 low
* , /	left	2 heigh

- *expr → expr + term | expr - term | term*
  *term → term * factor | term / factor | factor*
  factor → digit | (expr)
  digit → 0 | 1 | ... | 9

- Syntax of statements
  - stmt → id = expr ;
    | if (expr) stmt ;
    | if (expr) stmt else stmt ;
    | while (expr) stmt ;

    expr → expr + term | expr - term | term
    term → term * factor | term / factor | factor
    factor → digit | (expr)
    digit → 0 | 1 | … | 9

2.3 SYNTAX-DIRECTED TRANSLATION(SDT)

A formalism for specifying translations for programming language constructs.
(attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation

Postfix notation for an expression E

- If E is a variable or constant, then the postfix nation for E is E itself (E.t≡E).
- if E is an expression of the form E1 op E2 where op is a binary operator
  - E1' is the postfix of E1,
  - E2' is the postfix of E2
  - then E1' E2' op is the postfix for E1 op E2
- if E is (E1), and E1' is a postfix
  then E1' is the postfix for E

eg) 9 - 5 + 2 ⇒ 9 5 - 2 +

9 - (5 + 2) ⇒ 9 5 2 + -

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively for translation.
- A translation is an input-output mapping procedure for translation of an input X,
  - construct a parse tree for X.
  - synthesize attributes over the parse tree.

- Suppose a node n in parse tree is labeled by X and X.a denotes the value of attribute a of X at that node.
- compute X's attributes X.a using the semantic rules associated with X.

Example 2.6. SDD for infix to postfix translation

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
. . .	. . .
$term \rightarrow 9$	$term.t := '9'$

Fig 2.5. Syntax-directed definition for infix to postfix translation.

An example of synthesized attributes for input X=9-5+2

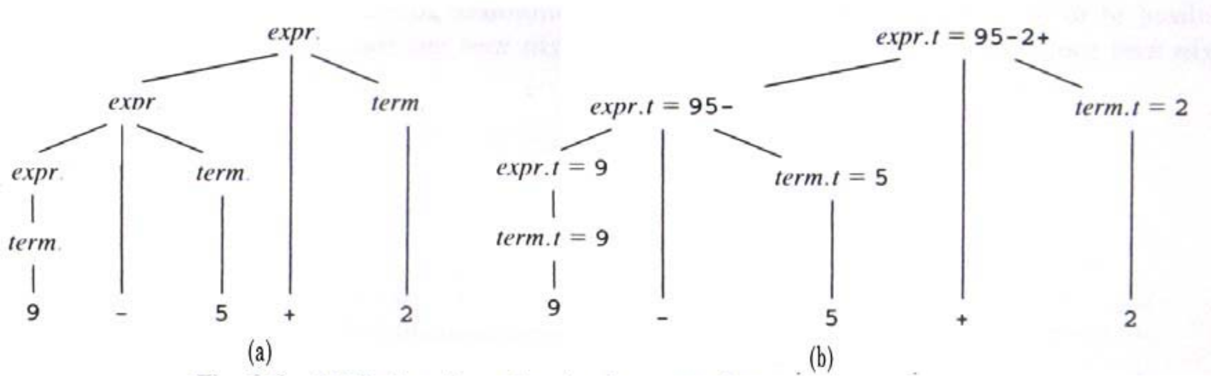


Fig 2.6. Attribute values at nodes in a parse tree.

Syntax-directed Translation Schemes(SDTS)

- A translation scheme is a context-free grammar in which program fragments called translation actions are embedded within the right sides of the production.

productions(postfix)	SDD for postfix to infix notation	SDTS
list → list + term	list.t = list.t || term.t || "+"	list → list + term {print("+")}

- {print("+");} : translation(semantic) action.
- SDTS generates an output for each sentence x generated by underlying grammar by executing actions in the order they appear during depth-first traversal of a parse tree for x.

1. Design translation schemes(SDTS) for translation
2. Translate :
   a) parse the input string x and
   b) emit the action result encountered during the depth-first traversal of parse tree.

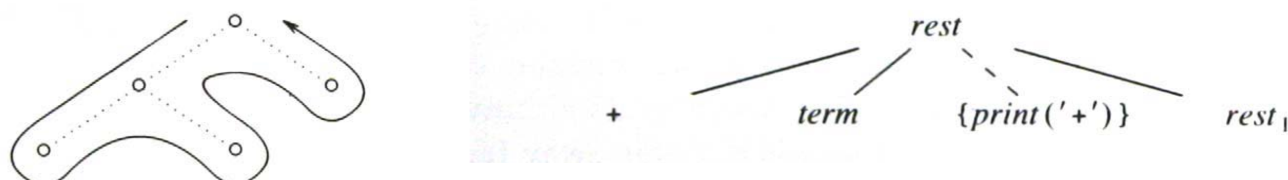


Fig 2.7. Example of a depth-first traversal of a tree. **Fig 2.8.** An extra leaf is constructed for a semantic action.

Example 2.8.

- SDD vs. SDTS for infix to postfix translation.

productions	SDD	SDTS
expr → list + term	expr.t = list.t || term.t || "+"	expr → list + term
expr → list + term	expr.t = list.t || term.t || "-"	printf{"+")}
expr → term	expr.t = term.t	expr → list + term printf{"-")}
term → 0	term.t = "0"	expr → term
term → 1	term.t = "1"	term → 0 printf{"0")}
…	…	term → 1 printf{"1")}
term → 9	term.t = "9"	…
		term → 9 printf{"0")}

- Action translating for input 9-5+2

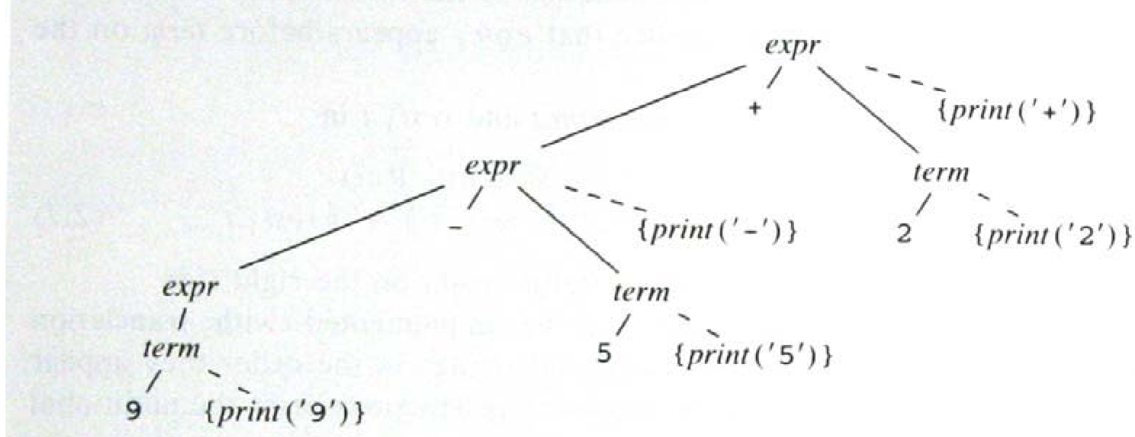


Fig 2.9. Actions translating 9-5+2 into 95-2+.

1) Parse.
2) Translate.

Do we have to maintain the whole parse tree ?

No, Semantic actions are performed during parsing, and we don't need the nodes (whose semantic actions done).

2.4 PARSING

if token string x ∈ L(G), then parse tree

else error message

Top-Down parsing

1. At node n labeled with nonterminal A, select one of the productions whose left part is A and construct children of node n with the symbols on the right side of that production.
2. Find the next node at which a sub-tree is to be constructed.

ex. G: type → simple

|↑id

|array [simple] of type

simple → integer

|char

|num dotdot num

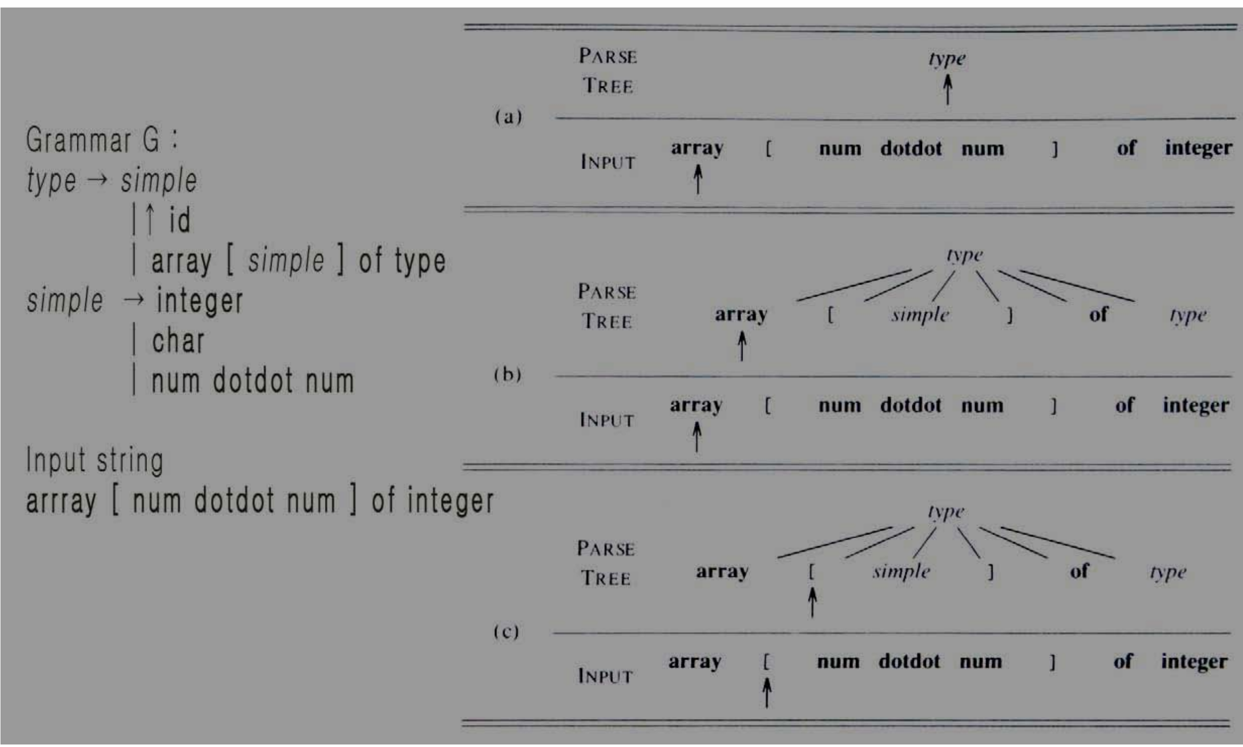


Fig 2.10. Top-down parsing while scanning the input from left to right.

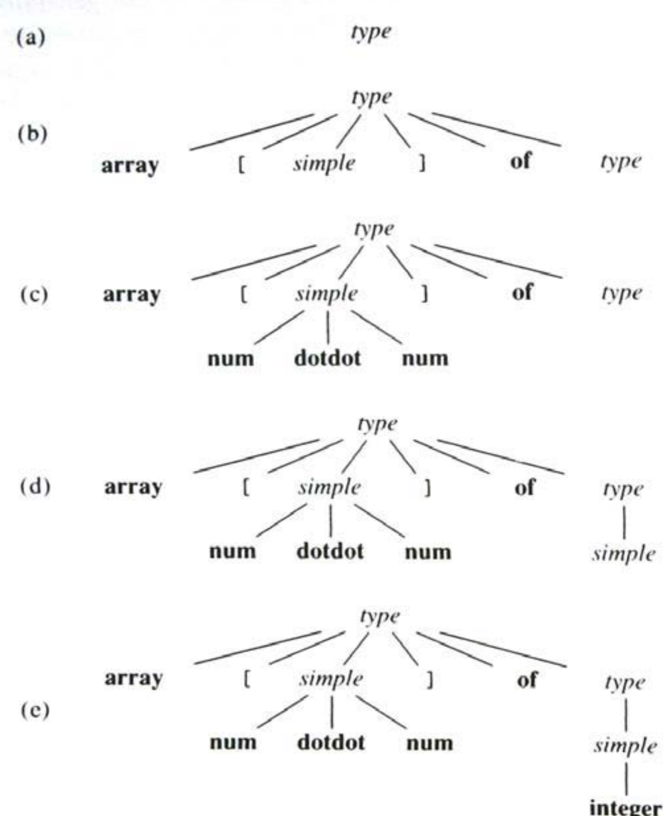


Fig 2.11. Steps in the top-down construction of a parse tree.

- The selection of production for a nonterminal may involve trial-and-error. => backtracking

- G : { S->aSb | c | ab }
  According to topdown parsing procedure, acb , aabb∈L(G)?
- S/acb⇒aSb/acb⇒aSb/acb⇒aaSbb/acb ⇒ X
  (S→aSb) move (S→aSb) backtracking
  ⇒aSb/acb⇒acb/acb⇒acb/acb⇒acb/acb
  (s→c) move move

  so, acb∈ L(G)
  Is is finished in 7 steps including one backtracking.

- S/aabb⇒aSb/aabb⇒aSb/aabb⇒aaSbb/aabb⇒aaSbb/aabb⇒aaaSbbb/aabb ⇒ X
  (S→aSb) move (S→aSb) move (S→aSb) backtracking
  ⇒aaSbb/aabb⇒aacbb/aabb ⇒ X
  (S→c) backtracking
  ⇒aaSbb/aabb⇒aaabbb/aabb⇒ X
  (S→ab) backtracking
  ⇒aaSbb/aabb⇒ X
  backtracking
  ⇒aSb/aabb⇒acb/aabb
  (S→c) bactracking
  ⇒aSb/aabb⇒aabb/aabb⇒aabb/aabb⇒aabb/aabb⇒aaba/aabb
  (S→ab) move move move

  so, aabb∈L(G)
  but process is too difficult. It needs 18 steps including 5 backtrackings.

- procedure of top-down parsing
  let a pointed grammar symbol and pointed input symbol be g, a respectively.
  - if(g ∈ N) select and expand a production whose left part equals to g next to current production.
    else if(g = a) then make g and a be a symbol next to current symbol.
    else if(g ≠a) back tracking
    - let the pointed input symbol a be the symbol that moves back to steps same with the number of current symbols of underlying production
    - eliminate the right side symbols of current production and let the pointed symbol g be the left side symbol of current production.

Predictive parsing (Recursive Decent Parsing,RDP)

- A strategy for the general top-down parsing
  Guess a production, see if it matches, if not, backtrack and try another.
  ⇒
- It may fail to recognize correct string in some grammar G and is tedious in processing.
  ⇒
- **Predictive parsing**
  - is a kind of top-down parsing that predicts a production whose derived terminal symbol is equal to next input symbol while expanding in top-down paring.
  - without backtracking.
  - Procedure decent parser is a kind of predictive parser that is implemented by disjoint recursive procedures one procedure for each nonterminal, the procedures are patterned after the productions.
- procedure of predictive parsing(RDP)
  let a pointed grammar symbol and pointed input symbol be g, a respectively.
  - if(g ∈ N)
    - select next production P whose left symbol equals to g and a set of first terminal symbols of derivation from the right symbols of the production P includes a input symbol a.
    - expand derivation with that production P.
  - else if(g = a) then make g and a be a symbol next to current symbol.
  - else if(g ≠a) error

- G : { S→aSb | c | ab } => G1 : { S->aS' | c S'->Sb | ab }
  According to predictive parsing procedure, acb , aabb∈L(G)?
  - S/acb⇒ confused in { S→aSb, S→ab }
  - so, a predictive parser requires some restriction in grammar, that is, there should be only one production whose left part of productions are A and each first terminal symbol of those productions have unique terminal symbol.
- Requirements for a grammar to be suitable for RDP: For each nonterminal either
  1. A → Bα, or
  2. A → a1α1 | a2α2 | … | anαn
     1) for 1 ≦ i, j ≦ n and i≠ j, ai ≠ aj
     2) A ε may also occur if none of ai can follow A in a derivation and if we have A→ε

- If the grammar is suitable, we can parse efficiently without backtrack.
  General top-down parser with backtracking
  ↓
  Recursive Descent Parser without backtracking
  ↓
  Picture Parsing (a kind of predictive parsing) without backtracking

Left Factoring

- If a grammar contains two productions of form
  S→ aα and S → aβ
  it is not suitable for top down parsing without backtracking. Troubles of this form can sometimes be removed from the grammar by a technique called the left factoring.
- In the left factoring, we replace { S→ aα, S→ aβ } by
  { S → aS', S'→ α, S'→ β } cf. S→ a(α|β)
  (Hopefully α and β start with different symbols)
- left factoring for G { S→aSb | c | ab }
  S→aS' | c   cf. S(=aSb | ab | c = a (Sb | b) | c) → a S' | c
  S'→Sb | b
- A concrete example:
  <stmt> →   IF <boolean> THEN <stmt> |
                        IF <boolean> THEN <stmt> ELSE <stmt>
  is transformed into
  <stmt>→   IF <boolean> THEN <stmt> S'
  S' →   ELSE <stmt> | ε

- **Example**,
  - for G1 : { S→aSb | c | ab }
    According to predictive parsing procedure, acb , aabb∈L(G)?
    - S/aabb⇒ unable to choose { S→aSb, S→ab ?}
  - According for the feft factored gtrammar G1, acb , aabb∈L(G)?
    G1 : { S→aS'|c S'→Sb|b} <= {S=a(Sb|b) | c }
  - S/acb⇒aS'/acb⇒aS'/acb ⇒ aSb/acb ⇒ acb/acb ⇒ acb/acb⇒ acb/acb
    (S→aS')   move   (S'→Sb⇒aS'b) (S'→c)   move   move
    so, acb∈ L(G)
    It needs only 6 steps whithout any backtracking.
    cf. General top-down parsing needs 7 steps and I backtracking.
  - S/aabb⇒aS'/aabb⇒aS'/aabb⇒aSb/aabb⇒aS'b/aabb⇒aaS'b/aabb⇒aabb/aabb⇒ ⇒
    (S→aS')   move   (S'→Sb⇒aS'b)   (S'→aS')   move   (S'→b)   move move
    so, aabb∈L(G)
    but, process is finished in 8 steps without any backtracking.
    cf. General top-down parsing needs 18 steps including 5 backtrackings.

Left Recursion

- A grammar is left recursive iff it contains a nonterminal A, such that
  A⇒+ Aα, where is any string.
  - Grammar {S→ Sα | c} is left recursive because of S⇒Sα
  - Grammar {S→ Aα, A→ Sb | c} is also left recursive because of S⇒Aα⇒ Sbα
- If a grammar is left recursive, you cannot build a predictive top down parser for it.

1) If a parser is trying to match S & S→Sα, it has no idea how many times S must be applied
2) Given a left recursive grammar, it is always possible to find another grammar that generates the same language and is not left recursive.
3) The resulting grammar might or might not be suitable for RDP.

- After this, if we need left factoring, it is not suitable for RDP.
- Right recursion: Special care/Harder than left recursion/SDT can handle.

Eliminating Left Recursion

Let G be S→ S A | A

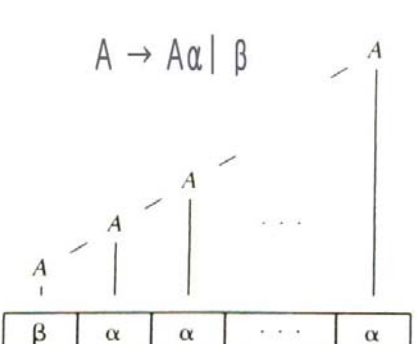
Note that a top-down parser cannot parse the grammar G, regardless of the order the productions are tried.

⇒ The productions generate strings of form AA⋯A

⇒ They can be replaced by S→A S' and S'→A S'| ε

Example :

- A → Aα| β
  =>
  A → βR
  R → αR | ε



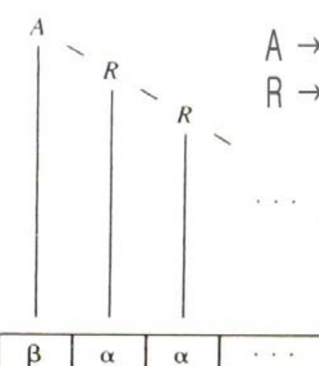


Fig 2.12. Left-and right-recursive ways of generating a string.

- In general, the rule is that
  - If A→ Aα1 | Aα2 | … | Aαn and
    A→ β1 | β2 | … | βm (no βi's start with A),
    then, replace by
    A → β1R | β2R| … | βmR and
    Z → α1R | α2R | … | αnR | ε

Exercise: Remove the left recursion in the following grammar:

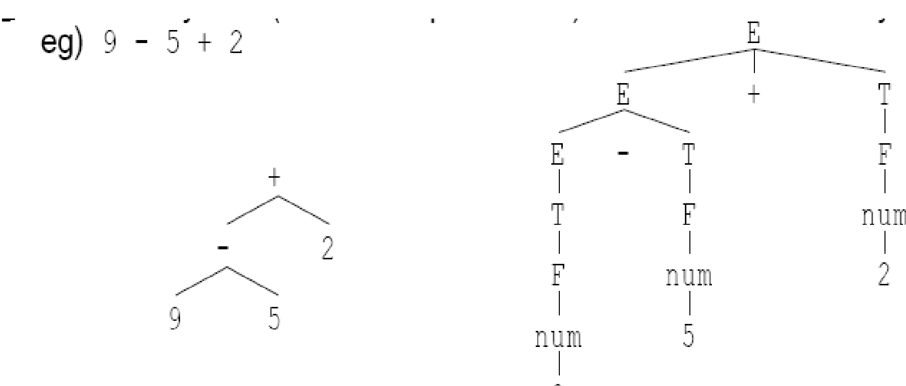
expr → expr + term | expr - term
expr → term

solution:

expr → term rest
rest → + term rest | - term rest | ε

2.5 A TRANSLATOR FOR SIMPLE EXPRESSIONS

- Convert infix into postfix(polish notation) using SDT.
- Abstract syntax (annotated parse tree) tree vs. Concrete syntax tree



- Concrete syntax tree : parse tree.
- Abstract syntax tree: syntax tree
- Concrete syntax : underlying grammar

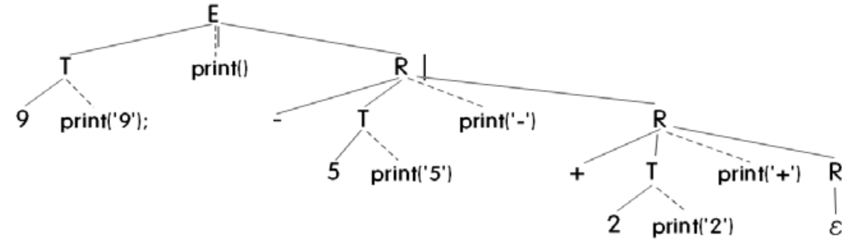
Adapting the Translation Scheme

- Embed the semantic action in the production
- Design a translation scheme
- Left recursion elimination and Left factoring
- Example

3) Design a translate scheme and eliminate left recursion

E→ E + T {'+'} E→ E - T {'-'} E→ T {} T→ 0{'0'}|…|9{'9'}	E→ T {} R R→ + T{'+'} R R→ - T{'-'} R R→ ε T→ 0{'0'}…|9{'9'}

4)Translate of a input string 9-5+2 : parsing and SDT



Result: 9 5 – 2 +

Example of translator design and execution

- A translation scheme and with left-recursion.

Initial specification for infix-to-postfix translator	with left recursion eliminated
expr → expr + term {printf{"+")} expr → expr - term {printf{"-")} expr → term term → 0 {printf{"0")} term → 1 {printf{"1")} … term → 9 {printf{"0")}	expr → term rest rest → + term {printf{"+")} rest rest → - term {printf{"-")} rest rest → ε term → 0 {printf{"0")} term → 1 {printf{"1")} … term → 9 {printf{"0")}

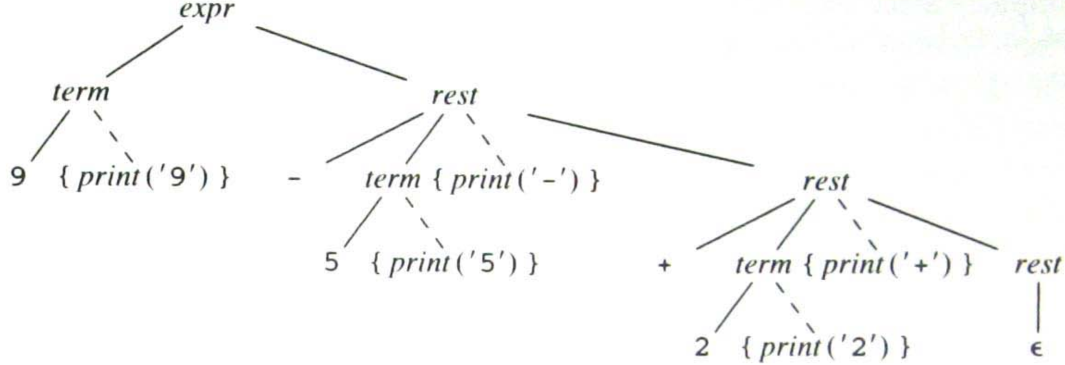


Fig 2.13. Translation of 9 – 5 +2 into 95-2+.

Procedure for the Nonterminal expr, term, and rest

```
expr()  //<expr → term rest>
{
      term(); rest();
}
rest()  //<rest → + term printf{"+")} rest | - term printf{"-")} rest | ε>
{
      if (lookahead == '+') {
          match('+'); term(); putchar('+'); rest();
      }
      else if (lookahead == '-') {
          match('-'); term(); putchar('-'); rest();
      }
      else ;
}
term() //< term → 0  printf{"0")} ··· term → 9  printf{"9")} >
{
      if (isdigit(lookahead)) {
          putchar(lookahead); match(lookahead);
      }
      else error();
}
```

Fig 2.14. Function for the nonterminals expr, rest, and term.

Optimizer and Translator

```
1. expr() {
2.     term(); rest();
3. }
4. rest()
5. {
6.     if(lookahead == '+' ) {
7.         m('+'); term(); p('+'); rest();
8.     } else  if(lookahead == '-' ) {
9.         m('-'); term(); p('-'); rest();
10.    } else ;
11. }
```

⇒

```
rest()
{
L: if(lookahead == '+' ) {
       m('+'); term(); p('+'); goto L;
   } else  if(lookahead == '-' ) {
       m('-'); term(); p('-'); goto L;
   } else ;
}
```

```
12. expr() {
13.     term();
14.     while(1) {
15.             if(lookahead == '+' ) {
16.         m('+'); term(); p('+');
17.     } else  if(lookahead == '-' ) {
18.         m('-'); term(); p('-');
19.     } else break;
20. }
```

2.6 LEXICAL ANALYSIS

- reads and converts the input into a stream of tokens to be analyzed by parser.
- lexeme : a sequence of characters which comprises a single token.
- Lexical Analyzer →Lexeme / Token → Parser

Removal of White Space and Comments

- Remove white space(blank, tab, new line etc.) and comments

Contsants

- Constants: For a while, consider only integers
- eg) for input 31 + 28, output(token representation)?
  input : 31 + 28
  output: <num, 31> <+, > <num, 28>
    num + :token
    31 28 : attribute, value(or lexeme) of integer token num

Recognizing

- Identifiers
    - Identifiers are names of variables, arrays, functions...
    - A grammar treats an identifier as a token.
    - eg) input : count = count + increment;
      output : <id,1> <=, > <id,1> <+, > <id, 2>;
      Symbol table

	tokens	attributes(lexeme)
0		
1	id	count
2	id	increment
3		

- Keywords are reserved, i.e., they cannot be used as identifiers.

Then a character string forms an identifier only if it is no a keyword.

- punctuation symbols
    - operators : + - * / := < > …

Interface to lexical analyzer

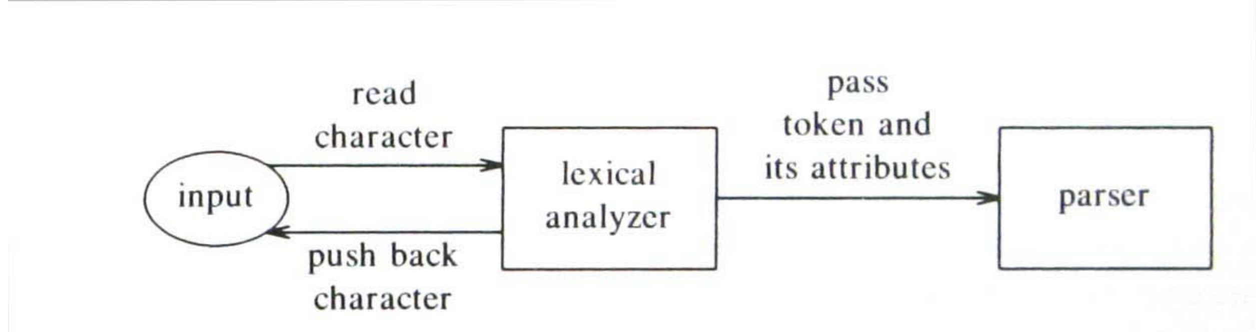


Fig 2.15. Inserting a lexical analyzer between the input and the parser

A Lexical Analyzer

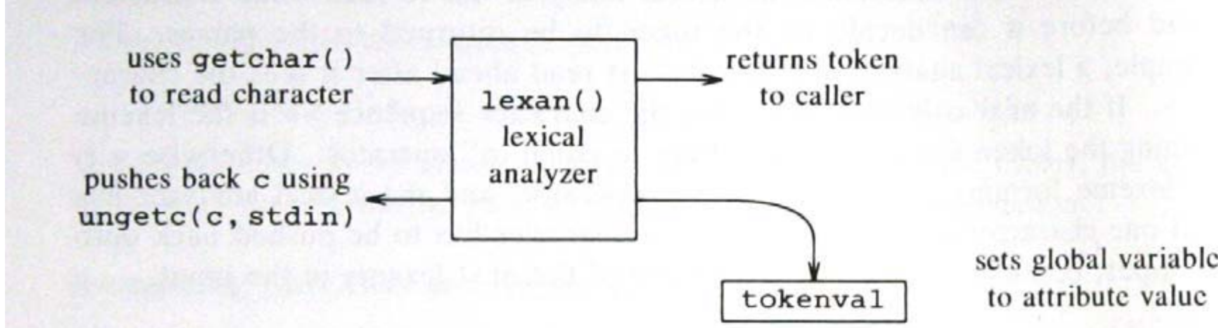


Fig 2.16. Implementing the interactions in Fig. 2.15.

- c=getchcar(); ungetc(c,stdin);
- token representation
    - #define NUM 256
- Function lexan()

  eg) input string 76 + a

  input , output(returned value)

  76 NUM, tokenval=76 (integer)

  \+ +

  A id , tokeval="a"

- A way that parser handles the token NUM returned by laxan()
    - consider a translation scheme

      factor → (expr)

      | num { print(num.value) }

      #define NUM 256

```
...
factor() {
        if(lookahead == '(' ) {
        match('('); exor(); match(')');
    } else if (lookahead == NUM) {
        printf(" %f ",tokenval); match(NUM);
    } else error();
}
```

- The implementation of function lexan

```
1)  #include <stdio.h>
2)  #include <ctype.h>
3)  int lino = 1;
4)  int tokenval = NONE;
5)  int lexan() {
6)          int t;
7)          while(1) {
8)                  t = getchar();
9)                  if ( t==' ' || t=='\t' ) ;
10)                 else if ( t=='\n' ) lineno +=1;
11)                 else if (isdigit(t)) {
12)                         tokenval = t -'0';
13)                         t = getchar();
14)                         while ( isdigit(t)) {
15)                                         tokenval = tokenval*10 + t - '0';
16)                                 t =getchar();
17)                                 }
18)                                 ungetc(t,stdin);
19)                                 retunr NUM;
20)                 } else {
21)                         tokenval = NONE;
22)                         return t;
23)                         }
24)                 }
25)         }
```

2.7 INCORPORATION A SYMBOL TABLE

- The symbol table interface, operation, usually called by parser.
    - insert(s,t): input s: lexeme
      t: token
      output index of new entry
    - lookup(s): input s: lexeme
      output index of the entry for string s, or 0 if s is not found in the symbol table.
- Handling reserved keywords
  1. Inserts all keywords in the symbol table in advance.
     ex) insert("div", div)

insert("mod", mod)

2. while parsing

- whenever an identifier s is encountered.
  if (lookup(s)'s token in {keywords}) s is for a keyword; else s is for a identifier;

- example
  - preset
    insert("div",div);
    insert("mod",mod);
  - while parsing
    lookup("count")=>0 insert("count",id);
    lookup("i") =>0 insert("i",id);
    lookup("i") =>4, id
    llokup("div")=>1,div

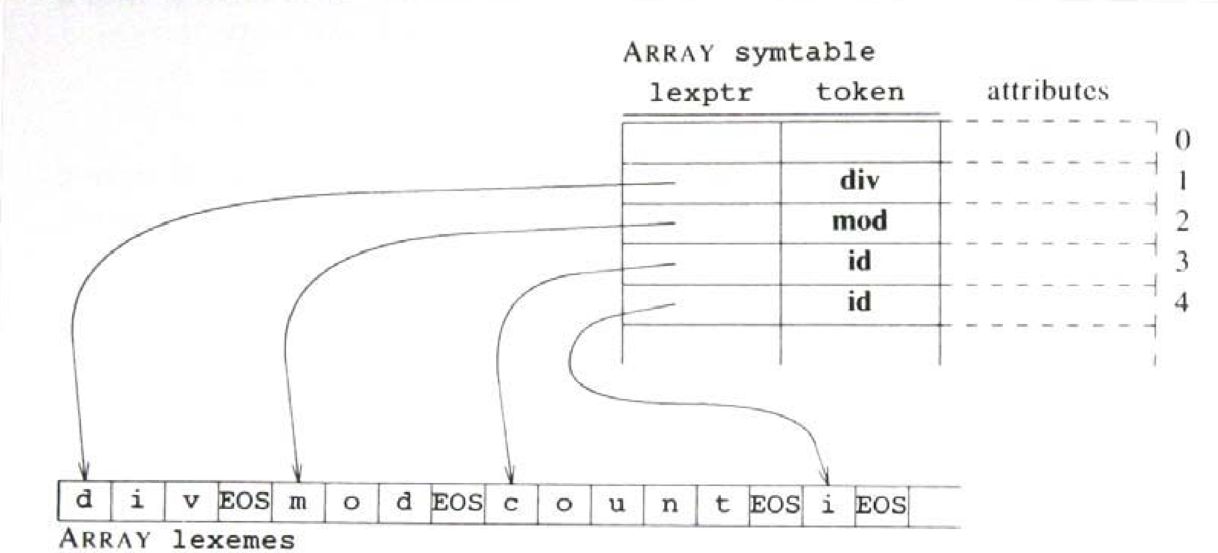


Fig 2.17. Symbol table and array for storing strings.

2.8 ABSTRACT STACK MACHINE

- An abstract machine is for intermediate code generation/execution.
- Instruction classes: arithmetic / stack manipulation / control flow

- 3 components of abstract stack machine
  1) Instruction memory : abstract machine code, intermediate code(instruction)
  2) Stack
  3) Data memory
- An example of stack machine operation.
  - for a input (5+a)*b, intermediate codes : push 5 rvalue 2 ....

instruction memory
push 5
rvalue 2
+
rvalue 3
*

Data memory

1	0	
2	11	a
3	7	b

Stack

1	2	3	4	5
5	11	16	7	112
	5		16	

L-value and r-value

- l-values a : address of location a
- r-values a : if a is location, then content of location a
  if a is constant, then value a
- eg) a :=5 + b;
  lvalue a⇒2 r value 5 ⇒ 5 r value of b ⇒ 7

Stack Manipulation

- Some instructions for assignment operation
  - push v : push v onto the stack.
  - rvalue a : push the contents of data location a.
  - lvalue a : push the address of data location a.
  - pop : throw away the top element of the stack.
  - := : assignment for the top 2 elements of the stack.
  - copy : push a copy of the top element of the stack.

Translation of Expressions

- Infix expression(IE) → SDD/SDTS → Abstact macine codes(ASC) of postfix expression for stack machine evaluation.
  eg)
  - IE: a + b, (⇒PE: a b +) ⇒ IC: rvalue a
    rvalue b
    +
  - day := (1461 * y) div 4 + (153 * m + 2) div 5 + d
    (⇒ day 1462 y * 4 div 153 m * 2 + 5 div + d + :=)
    ⇒ 1) lvalue day 6) div 11) push 5 16) :=
    2) push 1461 7) push 153 12) div
    3) rvalue y 8) rvalue m 13) +
    4) * 9) push 2 14) rvalue d
    5) push 4 10) + 15) +
- A translation scheme for assignment-statement into abstract astack machine code e can be expressed formally In the form as follows:
  stmt → id := expr
  { stmt.t := 'lvalue' || id.lexeme || expr.t || ':=' }
  eg) day :=a+b ⇒ lvalue day rvalue a rvalue b + :=

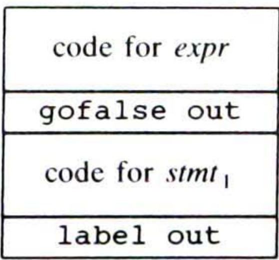
Control Flow

- 3 types of jump instructions :
    - Absolute target location
    - Relative target location(distance :Current ↔Target)
    - Symbolic target location(*i.e.* the machine supports labels)
- Control-flow instructions:
    - **label a**: the jump's target a
    - **goto a**: the next instruction is taken from statement labeled a
    - **gofalse a**: pop the top & if it is 0 then jump to a
    - **gotrue a**: pop the top & if it is nonzero then jump to a
    - **halt** : stop execution

Translation of Statements

- Translation scheme for translation if-statement into abstract machine code.
  stmt → if expr then stmt1
  { out := newlabel1)
  stmt.t := expr.t || 'gofalse' out || stmt1.t || 'label' out }

IF



WHILE

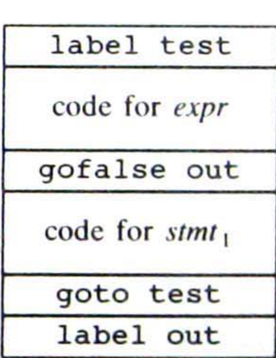


Fig 2.18. Code layout for conditional and while statements.

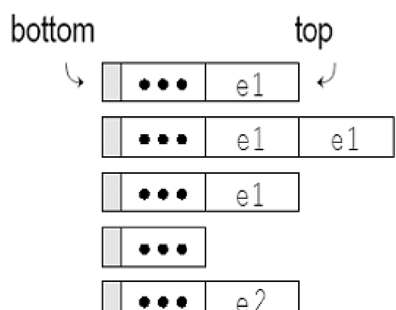
- Translation scheme for while-statement ?

Emitting a Translation

- Semantic Action(Tranaslation Scheme):
  1. stmt → if
     expr { out := newlabel; emit('gofalse', out) }
     then
     stmt1 { emit('label', out) }
  2. stmt → id { emit('lvalue', id.lexeme) }
     :=
     expr { emit(':=') }
  3. stmt → i
     expr { out := newlabel; emit('gofalse', out) }
     then
     stmt1 { emit('label', out) ; out1 := newlabel; emit('goto', out`1); }

else
stmt2 { emit('label', out1) ; }
if(expr==false) goto out
stmt1 goto out1
out : stmt2
out1:

Implementation



- procedure stmt()
- var test,out:integer;
- begin
  - if lookahead = id then begin
    - emit('lvalue',tokenval); match(id); match(':='); expr(); emit(':=');
  - end
  - else if lookahead = 'if' then begin
    - match('if');
    - expr();
    - out := newlabel();
    - emit('gofalse', out);
    - match('then');
    - stmt;
    - emit('label', out)
  - end
  - else error();
- end

Control Flow with Analysis

- if E1 or E2 then S vs if E1 and E2 then S
  E1 or E2 = if E1 then true else E2
  E1 and E2 = if E1 then E2 else false
- **The code for E1 or E2.**
  - Codes for E1 Evaluation result: e1
  - copy
  - gotrue OUT
  - pop
  - Codes for E2 Evaluation result: e2
  - label OUT

- The full code for **if E1 or E2 then S ;**
  - codes for E1
  - copy
  - gotrue OUT1
  - pop
  - codes for E2
  - label OUT1

- o gofalse OUT2
  - o code for S
  - o label OUT2
- Exercise: How about if E1 and E2 then S;
  - o if E1 and E2 then S1 else S2;

2.9 Putting the techniques together!

- infix expression ⇒ postfix expression
  eg) id+(id-id)*num/id ⇒ id id id - num * id / +

Description of the Translator

- Syntax directed translation scheme (SDTS) to translate the infix expressions into the postfix expressions,

```
start  → list eof
list   → expr ; list
       | ε
expr   → expr + term        { print('+') }
       | expr - term        { print('-') }
       | term
term   → term * factor      { print('*') }
       | term / factor      { print('/') }
       | term div factor    { print('DIV') }
       | term mod factor    { print('MOD') }
       | factor
factor → ( expr )
       | id                 { print(id.lexeme) }
       | num                { print(num.value) }
```

Fig 2.19. Specification for infix-to-postfix translation

Structure of the translator,

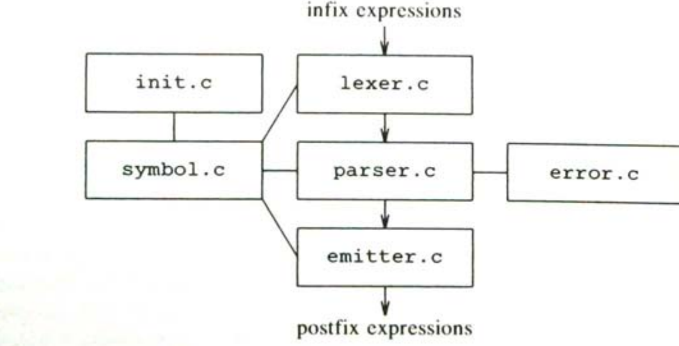


Fig 2.19. Modules of infix to postfix translator.

- o global header file "header.h"

The Lexical Analysis Module lexer.c

- o Description of tokens
  + - * / DIV MOD ( ) ID NUM DONE

LEXEME	TOKEN	ATTRIBUTE VALUE
white space		
sequence of digits	NUM	numeric value of sequence
div	DIV	
mod	MOD	
other sequences of a letter then letters and digits	ID	index into symtable
end-of-file character	DONE	
any other character	that character	NONE

Fig 2.20. Description of tokens.

The Parser Module parser.c

SDTS
|| ← left recursion elimination
New SDTS

```
start → list eof
 list → expr ; list
      | ε
 expr → expr + term        { print('+') }
      | expr - term        { print('-') }
      | term

 term → term * factor      { print('*') }
      | term / factor      { print('/') }
      | term div factor    { print('DIV') }
      | term mod factor    { print('MOD') }
      | factor

factor → ( expr )
      | id                 { print(id.lexeme) }
      | num                { print(num.value) }
```

```
    start → list eof
     list → expr ; list
          | ε
     expr → term moreexpr
 moreexpr → + term { print('+') } moreexpr
          | - term { print('-') } moreexpr
          | ε
     term → factor moreterm
 moreterm → * factor { print('*') } moreterm
          | / factor { print('/') } moreterm
          | div factor { print('DIV') } moreterm
          | mod factor { print('MOD') } moreterm
          | ε
   factor → ( expr )
          | id            { print(id.lexeme) }
          | num           { print(num.value) }
```

Fig 2.20. Specification for infix to postfix translator & syntax directed translation scheme after eliminating left-recursion.

The Emitter Module emitter.c

emit (t,tval)

The Symbol-Table Modules symbol.c and init.c

Symbol.c

data structure of symbol table Fig 2.29 p62

insert(s,t)

lookup(s)

The Error Module error.c

Example of execution

input 12 div 5 + 2

output 12

5

div

2

+

3. Lexical Analysis:

3.1 OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

3.2 ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

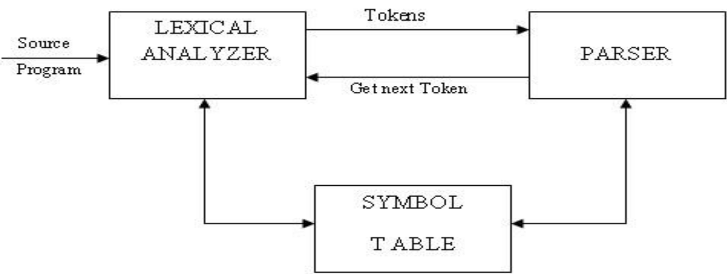


Fig. 3.1: Role of Lexical analyzer

Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

3.3 TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,
1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants
Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Token	lexeme	pattern
const	const	const
if	if	If
relation	<,<=,= ,<>,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w "and "except"
literal	"core"	pattern

Fig. 3.2: Example of Token, Lexeme and Pattern

3.4. LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

i. Delete one character from the remaining input.
ii. Insert a missing character in to the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

3.5. REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string. Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R1|R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each 'a' in Σ, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

 (R) | (S) means L(r) U L(s)
 R.S means L(r).L(s)
 R* denotes L(r*)

3.6. REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.
Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

 Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

 Letter - A | B | ……| Z | a | b |……| z|
 Digits - 0 | 1 | 2 | …. | 9
 Id - letter (letter / digit)*

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

 Stmt →if expr then stmt
  | If expr then else stmt
  | ϵ
 Expr →term relop term
  | term
 Term →id
  |number

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes.
The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

 digit → [0,9]
 digits →digit+
 number →digit(.digit)?(e.[+-]?digits)?
 letter → [A-Z,a-z]
 id →letter(letter/digit)*
 if → if
 then →then

else →else
relop →< | > |<= | >= | = = | < >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

WS → (blank/tab/newline)+

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any WS	-	-
if	if	-
then	then	-
else	else	-
Any id	Id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
==	relop	EQ
<>	relop	NE

3.7. TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.

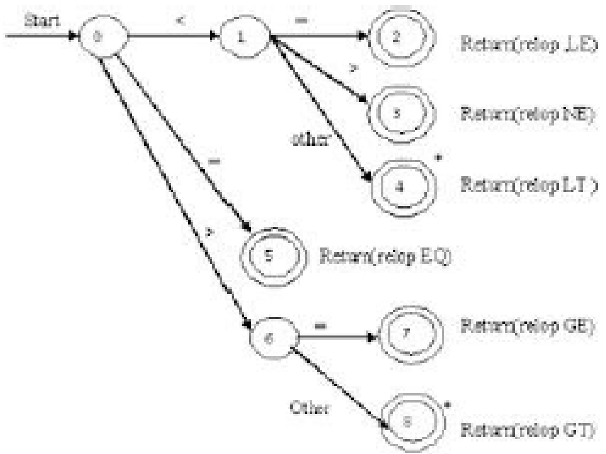


Fig. 3.3: Transition diagram of Relational operators

As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.

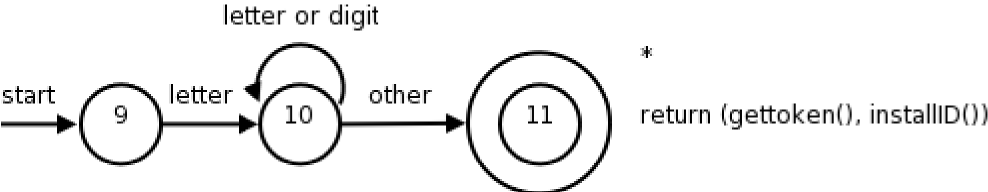


Fig. 3.4: Transition diagram of Identifier

The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

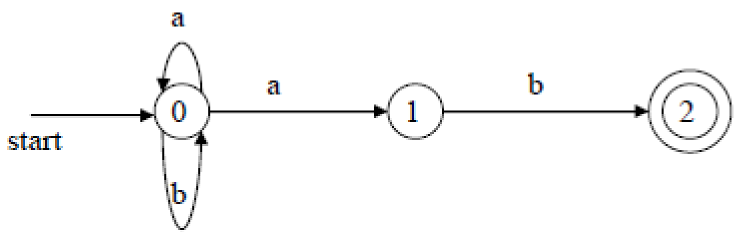
3.8. FINITE AUTOMATON

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

3.9. Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move - a transition function move to map state-symbol pairs to sets of states.
  - s0 - a start (initial) state
  - F- a set of accepting states (final states)
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Example:



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

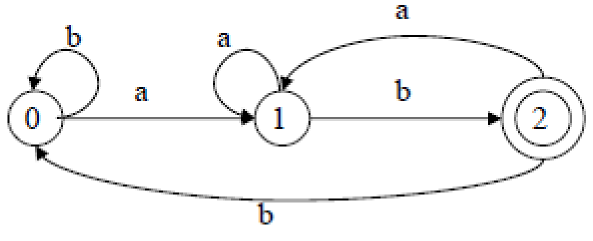
The language recognized by this NFA is (a|b)*ab

3.10. Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has ε- transition
- For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transition function is from pair of state-symbol to state (not set of states)

Example:

The DFA to recognize the language (a|b)* ab is as follows.



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

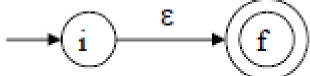
Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

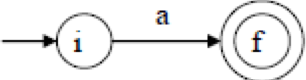
Note that the entries in this function are single value and not set of values (unlike NFA).

3.11. Converting RE to NFA

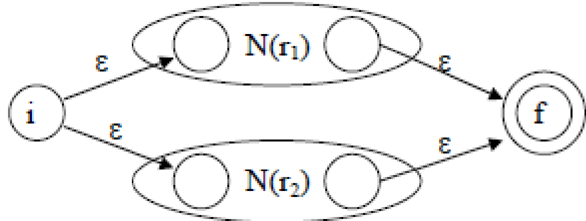
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.
- To recognize an empty string ε:



- To recognize a symbol a in the alphabet Σ :

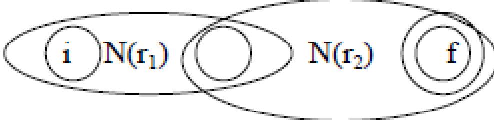


- For regular expression r1 | r2:



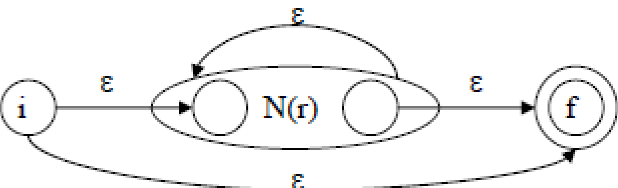
N(r1) and N(r2) are NFAs for regular expressions r1 and r2.

- For regular expression r1 r2

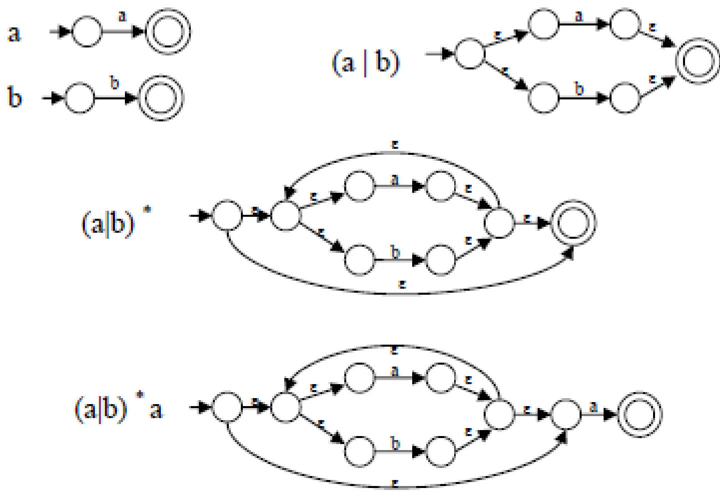


Here, final state of N(r1) becomes the final state of N(r1r2).

- For regular expression r*



Example:
For a RE (a|b) * a, the NFA construction is shown below.



3.12. Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an –transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an -transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The **-closure** function takes a state and returns the set of states reachable from it based on (one or more) -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its -closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

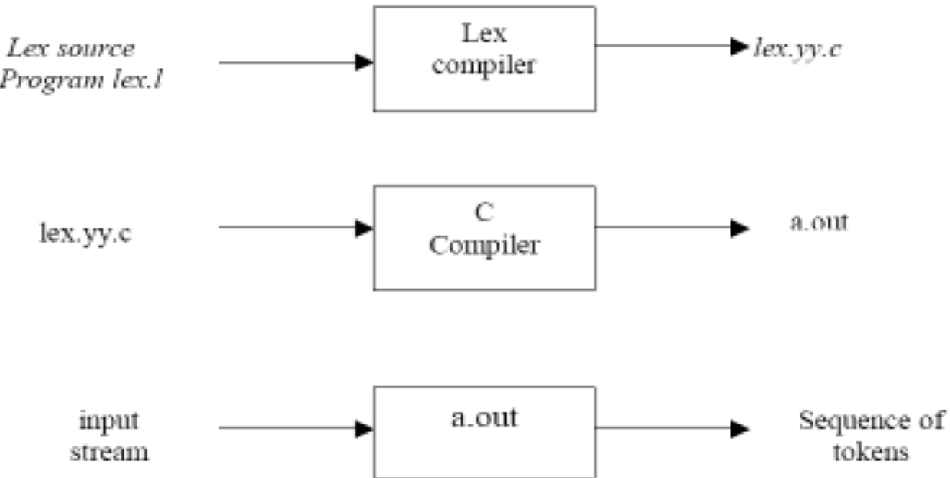
We can generalise both these functions to apply to sets of states by taking the union of the application to individual states.

For Example, if A, B and C are states, move({A,B,C},`a') = move(A,`a') move(B,`a') move(C,`a').
The Subset Construction Algorithm is a follows:

```
put ε-closure({s0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
        begin
                mark S1
                for each input symbol a do
                        begin
                                S2 ← ε-closure(move(S1,a))
                                if (S2 is not in DS) then
                                add S2 into DS as an unmarked state
                                transfunc[S1,a] ← S2
                        end
        end
```

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure({s0})

3.13. Lexical Analyzer Generator



3.18. Lex specifications:
A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. # *define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

   p1 {action 1}
   p2 {action 2}
   p3 {action 3}
   ... ...
   ... ...

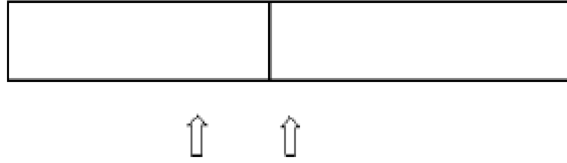
   Where, each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.
3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

3.19. INPUT BUFFERING

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
Buffering techniques:
1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see:

DECALRE (ARG1, ARG2… ARG *n*) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

SYNTAX ANALYSIS

4.1 ROLE OF THE PARSER :

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementry sutree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed are:

a. Top down parser: which build parse trees from top(root) to bottom(leaves)
b. Bottom up parser: which build parse trees from leaves and work up the root.

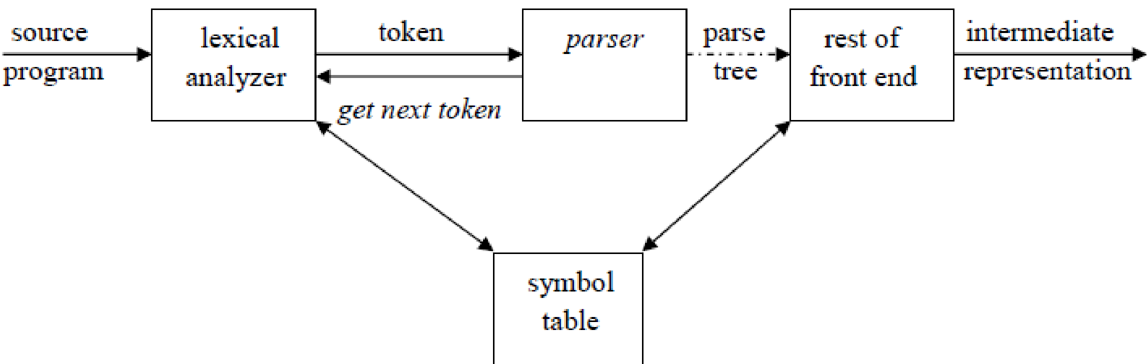


Fig . 4.1: position of parser in compiler model.

4.2 CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S).

Here , V is finite set of terminals (in our case, this will be the set of tokens)

T is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)

S is a start symbol (one of the non-terminal symbol)

L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff $S \Rightarrow \omega$ where ω is a string of terminals of G. If G is a context-free grammar, L(G) is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

Consider the production of the form $S \Rightarrow \alpha$, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

4.2.1 Derivations

In general a derivation step is

$\alpha A\beta \Rightarrow \alpha\gamma\beta$ is sentential form and if there is a production rule $A \rightarrow \gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha 1 \Rightarrow \alpha 2 \Rightarrow \ldots \Rightarrow \alpha n$ (αn derives from $\alpha 1$ or $\alpha 1$ derives αn). There are two types of derivaion

1 At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

2 If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:
E → E + E | E – E | E * E | E / E | - E
E → (E)
E → id

Leftmost derivation :

E → E + E

→ E * E+E →id* E+E→id*id+E→id*id+id

The string is derive from the grammar w= id*id+id, which is consists of all terminal symbols

Rightmost derivation

E → E + E

→ E+E * E→E+ E*id→E+id*id→id+id*id

Given grammar G : E → E+E | E*E | (E) | - E | id

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

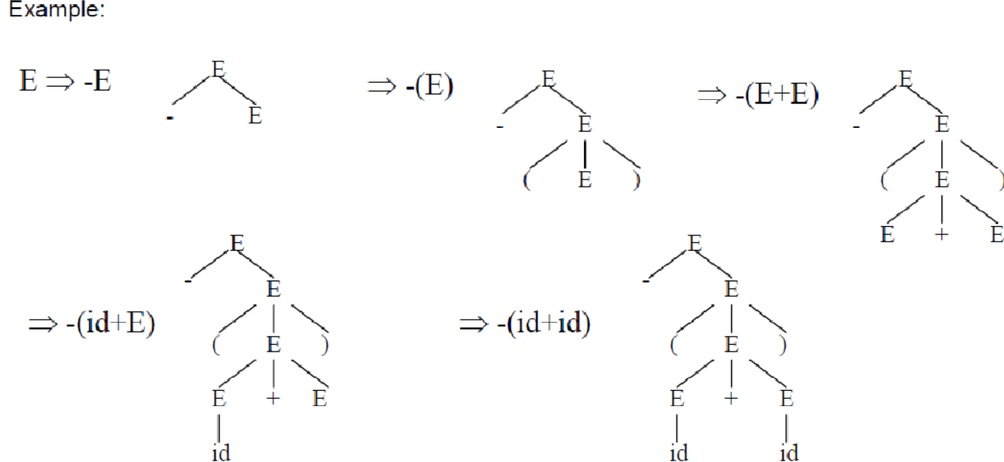
- Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

- Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

4.2.2 PARSE TREE

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.



Ambiguity:

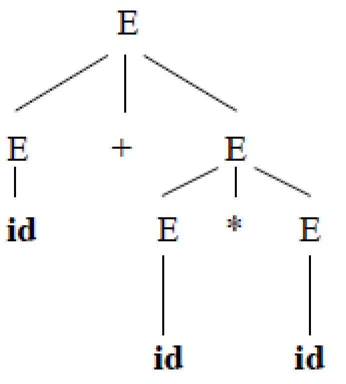
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

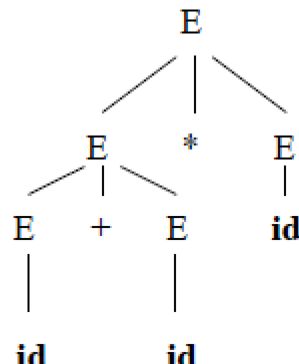
Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :





Example:

To disambiguate the grammar E → E+E | E*E | E^E | id | (E), we can use precedence of operators as follows:

^ (right to left)

/,* (left to right)

-,+ (left to right)

We get the following unambiguous grammar:

E → E+T | T

T → T*F | F

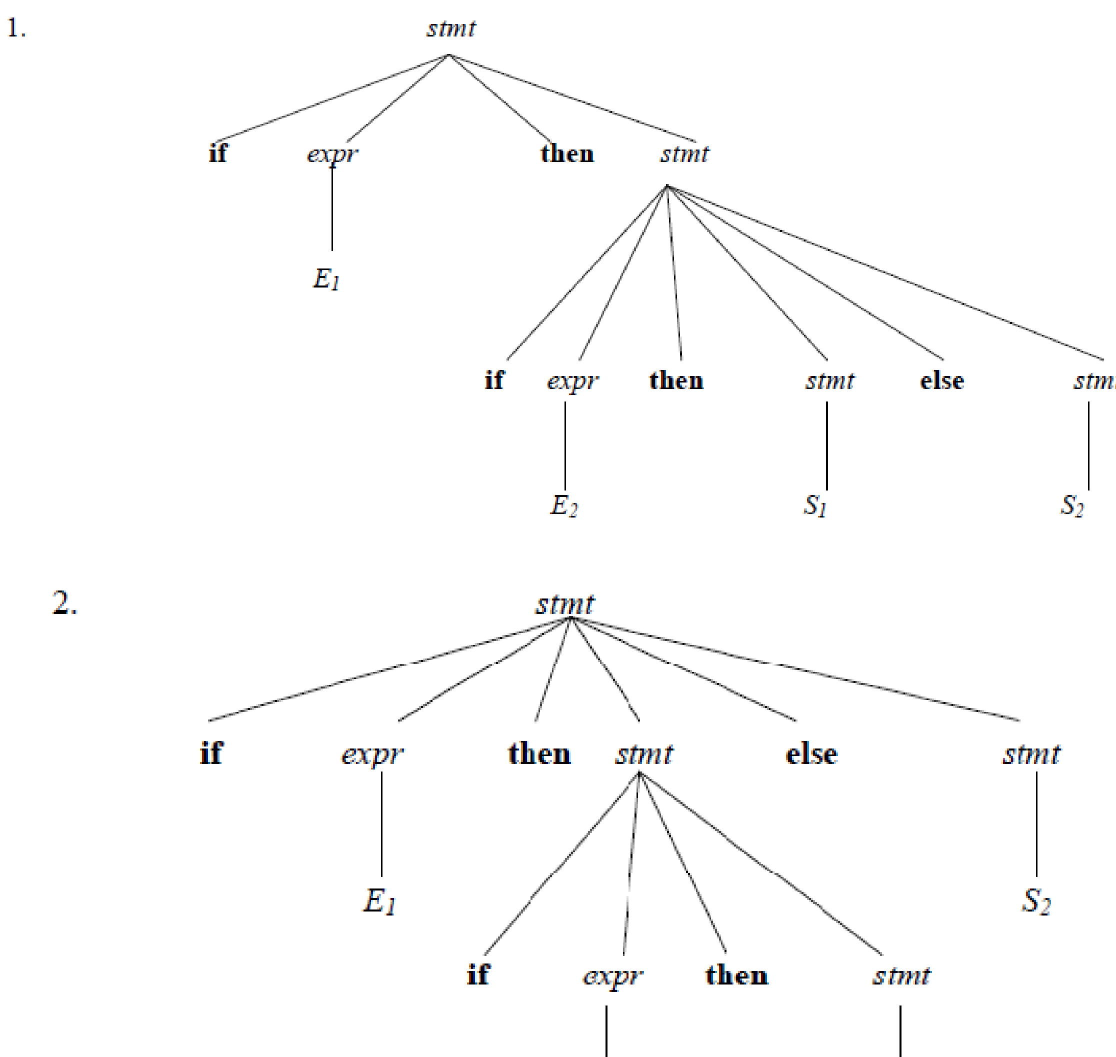
F → G^F | G

G → id | (E)

Consider this example, G: *stmt* → **if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following

Two parse trees for leftmost derivation :



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt* | **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

First eliminate the left recursion for E as

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

Then eliminate for T as

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

Thus the obtained grammar after eliminating left recursion is

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order A1, A2 . . . An.
2. **for** *i* := 1 **to** *n* **do begin**

    for *j* := 1 **to** *i*-1 **do begin**

        replace each production of the form $Ai \rightarrow Aj\ \gamma$

        by the productions $Ai \rightarrow \delta 1\ \gamma \mid \delta 2\gamma \mid \ldots \mid \delta k\ \gamma$

        where $Aj \rightarrow \delta 1 \mid \delta 2 \mid \ldots \mid \delta k$ are all the current Aj-productions;

    end

        eliminate the immediate left recursion among the Ai-productions

end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → αβ1 | αβ2 , it can be rewritten as

A → αA'

A' → β1 | β2

Consider the grammar , G : S → iEtS | iEtSeS | a

E → b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS | ε

E → b

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar G : S → cAd

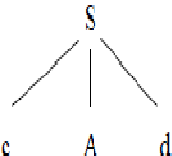
A → ab | a

and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

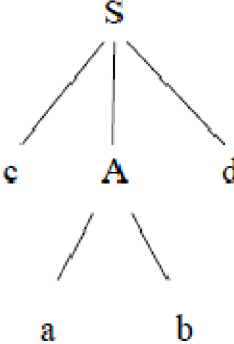
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



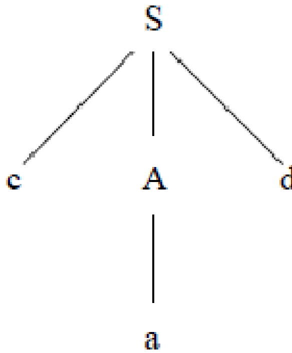
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.**

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

Recursive procedure:

```
Procedure E()
begin
        T( );
        EPRIME( );
End
Procedure EPRIME( )
        begin
                If input_symbol='+' then
                ADVANCE( );
                T( );
                EPRIME( );
        end
Procedure T( )
        begin
                F( );
                TPRIME( );
        End
```

```
Procedure TPRIME( )
    begin
        If input_symbol='*' then
        ADVANCE( );
        F( );
        TPRIME( );
    end
Procedure F( )
    begin
        If input-symbol='id' then
        ADVANCE( );
        else if input-symbol='(' then
        ADVANCE( );
        E( );
        else if input-symbol=')' then
        ADVANCE( );
    end
    else ERROR( );
```

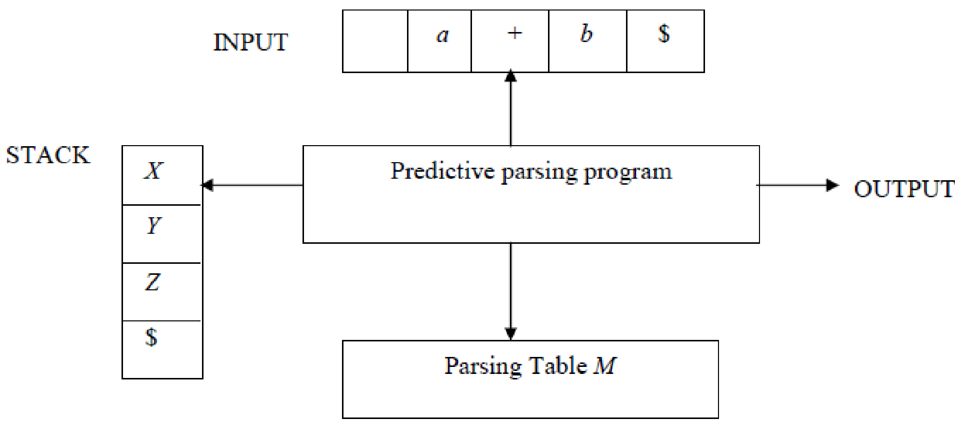
Stack implementation:

PROCEDURE	INPUT STRING
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id**+**id*id
TPRIME( )	id**+**id*id
EPRIME( )	id**+**id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id*****id
TPRIME( )	id+id*****id
ADVANCE( )	id+id*****id
F( )	id+id*****id
ADVANCE( )	id+id***id**
TPRIME( )	id+id***id**

2. PREDICTIVE PARSING

- ✓ Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- ✓ The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack:

It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table:

It is a two-dimensional array $M[A, a]$, where **'*A*'** is a non-terminal and **'*a*'** is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.

If $M[X, a] = \{X \rightarrow UVW\}$,the parser replaces X on top of the stack by UVW

If $M[X, a]$ = **error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in $L(G)$, a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has $\$S$ on the stack with *S*, the start symbol of *G* on top, and $w\$$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

set *ip* to point to the first symbol of $w\$$;

repeat

let *X* be the top stack symbol and *a* the symbol pointed to by *ip*;

if *X* is a terminal or $ **then**

if $X = a$ **then**

pop *X* from the stack and advance *ip*

else e*rror*()

else /* *X* is a non-terminal */

if $M[X, a] = X \rightarrow Y1Y2 \ldots Yk$ **then begin**

pop *X* from the stack;

push $Yk, Yk\text{-}1, \ldots, Y1$ onto the stack, with $Y1$ on top;

output the production $X \rightarrow Y1\ Y2 \ldots Yk$

end

else *error*()

until $X = \$$

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

Rules for first( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and $X \rightarrow a\alpha$ is a production then add *a* to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1\ Y_2 \ldots Y_k$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*),…,FIRST(*Yi-1*); that is, *Y1*,….*Yi-1* => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(*X*).

Rules for follow( ):

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production *A* → α*B*β, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method :

1. For each production *A* → α of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add *A* → α to *M*[*A*, *a*].
3. If ε is in FIRST(α), add A → α to *M*[*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M*[*A*, $].
4. Make each undefined entry of *M* be **error**.

Example:

Consider the following grammar :

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) = {+, * , $,) }

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E'→ ε
T	T → FT'			T → FT'		
T'		T'→ ε	T'→ *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.
Consider this following grammar:
S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have

S → iEtSS' | a

S'→ eS | ε

E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) = { i, a }

FIRST(S') = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S') = { $,e }

FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)		**RIGHTMOST DERIVATION**
a**b**bcde	(A → b)	**S** → aA**B**e
a**Abc**de	(A → Abc)	→ a**A**de
aA**d**e	(B → d)	→ a**A**bcde
aABe	(S → aABe)	→ abbcde
S		

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

E → **E+E**
→ E+**E*E**
→ E+E***id_3**
→ E+**id_2***id_3
→ **id_1**+id_2*id_3

In the above derivation the underlined substrings are called **handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the nth right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	id_1+id_2*id_3 $	shift
$ id_1	+id_2*id_3 $	reduce by E→id
$ E	+id_2*id_3 $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	*id_3 $	reduce by E→id
$ E+E	*id_3 $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. Reduce-reduce conflict:

Consider the grammar:

M → R+R | R+c | R
R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Viable prefixes:

- α is a viable prefix of the grammar if there is *w* such that α*w* is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ε or has two adjacent non-terminals.

Example:

Consider the grammar:

$E \rightarrow EAE \mid (E) \mid -E \mid id$
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid -E \mid id$

Operator precedence relations:

There are three disjoint precedence relations namely

$<\cdot$ - less than
$=$ - equal to
$\cdot>$ - greater than

The relations give the following meaning:

$a <\cdot b$ – a yields precedence to b
$a = b$ – a has the same precedence as b
$a \cdot> b$ – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$

2. If operators θ_1 and θ_2, are of equal precedence, then make
   $\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$ if operators are left associative
   $\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$ if right associative

3. Make the following for all operators θ:
   $\theta <\cdot id$, $id \cdot> \theta$
   $\theta <\cdot ($, $(<\cdot \theta$
   $) \cdot> \theta$, $\theta \cdot>)$
   $\theta \cdot> \$$, $\$ <\cdot \theta$

Also make
(=) , (<· (,) ·>) , (<· id , id ·>) , $ <· id , id ·> $, $ <· (,) ·> $

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

1. ↑ is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Operator precedence parsing algorithm:

Input : An input string *w* and a table of precedence relations.
Output : If *w* is well formed, a *skeletal* parse tree **,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.
Method : Initially the stack contains $ and the input buffer the string *w* $. To parse, we execute the following program :

(1) Set *ip* to point to the first symbol of *w*$;
(2) **repeat forever**
(3) **if** $ is on top of the stack and *ip* points to $ **then**
(4) **return**
else begin
(5) let *a* be the topmost terminal symbol on the stack
and let *b* be the symbol pointed to by *ip;*
(6) **if** *a* <· *b* or *a* = *b* **then begin**
(7) push *b* onto the stack;
(8) advance *ip* to the next input symbol;
end;

(9) **else if** *a* ·> *b* **then** /*reduce*/
(10) **repeat**
(11) pop the stack
(12) **until** the top stack terminal is related by <·
to the terminal most recently popped
(13) **else** error( )
end

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is **id+id*id** .The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

LR PARSERS

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

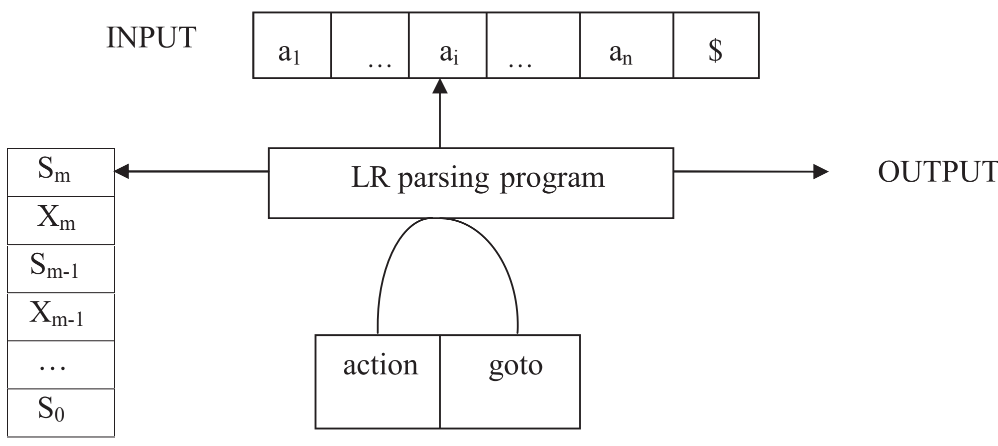
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



It consists of : an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults *action*$[s_m,a_i]$ in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever  begin
          let s be the state on top of the stack and
               a  the symbol pointed to by ip;
          if action[s, a] = shift s' then begin
               push a then s' on top of the stack;
               advance ip to the next input symbol
          end
          else if action[s, a] = reduce A→β then begin
                 pop 2* | β | symbols off the stack;
                 let s' be the state now on top of the stack;
                 push A then goto[s', A] on top of the stack;
                 output  the production A→ β
          end
          else if action[s, a] = accept then
                 return
          else error( )
end
```

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:
1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'
Output : The SLR parsing table functions *action* and *goto* for G'
Method :
1. Construct C = {$I_0, I_1, \ldots I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→α·] is in I_i , then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.

I_0 : E' → **.** E
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

GOTO (I_0 , E)
I_1 : E' → E **.**
E → E **.** + T

GOTO (I_4 , id)
I_5 : F → id **.**

GOTO (I_0 , T)

I_2 : E → T .

T → T . * F

GOTO (I_0 , F)

I_3 : T → F .

GOTO (I_0 , ( )

I_4 : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I_0 , id)

I_5 : F → id .

GOTO (I_1 , +)

I_6 : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I_2 , *)

I_7 : T → T * . F

F → . (E)

F → . id

GOTO (I_4 , E)

I_8 : F → (E .)

E → E . + T

GOTO (I_4 , T)

I_2 : E → T .

T → T . * F

GOTO (I_4 , F)

I_3 : T → F .

GOTO (I_6 , T)

I_9 : E → E + T .

T → T . * F

GOTO (I_6 , F)

I_3 : T → F .

GOTO (I_6 , ( )

I_4 : F → (. E)

GOTO (I_6 , id)

I_5 : F → id .

GOTO (I_7 , F)

I_{10} : T → T * F .

GOTO (I_7 , ( )

I_4 : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I_7 , id)

I_5 : F → id .

GOTO (I_8 ,))

I_{11} : F → (E) .

GOTO (I_8 , +)

I_6 : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I_9 , *)

I_7 : T → T * . F

F → . (E)

F → . id

GOTO (I_4 , ( )

I_4 : F $\rightarrow$ (. E)

E $\rightarrow$. E + T

E $\rightarrow$. T

T $\rightarrow$. T * F

T $\rightarrow$. F

F $\rightarrow$. (E)

F $\rightarrow$ id

FOLLOW (E) = { $,) , +)

FOLLOW (T) = { $, + ,) , * }

FOOLOW (F) = { * , + ,) , $ }

SLR parsing table:

	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ; **shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ; **reduce** by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ; **reduce** by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ; **reduce** by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ; **shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ; **shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ; **reduce** by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ; **reduce** by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ; **shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ; **shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ; **reduce** by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ; **reduce** by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ; **reduce** by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) = **accept**

MODULE 2 - SYNTAX-DIRECTED TRANSLATION

SYNTAX-DIRECTED TRANSLATION

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - In fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - A string, a number, a memory location, a complex record.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Example:

Production	Semantic Rule	Program Fragment
L → E **return**	print(E.val)	print(val[top-1])
E → E^1 + T	E.val = E^1.val + T.val	val[ntop] = val[top-2] + val[top]
E → T	E.val = T.val	
T → T^1 * F	T.val = T^1.val * F.val	val[ntop] = val[top-2] * val[top]
T → F	T.val = F.val	
F → (E)	F.val = E.val	val[ntop] = val[top-1]
F → **digit**	F.val = **digit**.lexval	val[top] = digit.lexval

- Symbols E, T, and F are associated with an attribute *val*.
- The token **digit** has an attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- The *Program Fragment* above represents the implementation of the semantic rule for a bottom-up parser.
- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).
- The above model is suited for a desk calculator where the purpose is to evaluate and to generate code.

Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.

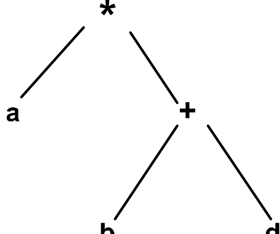
Syntax Tree

Syntax Tree is a variant of the Parse tree, where each leaf represents an operand and each interior node an operator.

Example:

Production	**Semantic Rule**
E → E1 **op** E2	E.val = NODE (op, E1.val, E2.val)
E → (E1)	E.val = E1.val
E → - E1	E.val = UNARY (- , E1.val)
E → **id**	E.val = LEAF (id)

A sentence **a*(b+d)** would have the following syntax tree:



Postfix Notation

Postfix Notation is another useful form of intermediate code if the language is mostly expressions.

Example:

Production	**Semantic Rule**	**Program Fragment**
E → E1 **op** E2	E.code = E1.code || E2.code || op	print op
E → (E1)	E.code = E1.code	
E → **id**	E.code = id	print id

Three Address Code

- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).
- The most general kind of three-address code is:

  x := y ***op*** z

  where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.
- But we may also the following notation for quadraples (much better notation because it looks like a machine code instruction)

  op y,z,x

  apply operator op to y and z, and store the result in x.

Representation of three-address codes

Three-address code can be represented in various forms viz. Quadruples, Triples and Indirect Triples. These forms are demonstrated by way of an example below.

Example:

A = -B * (C + D)

Three-Address code is as follows:

T1 = -B
T2 = C + D
T3 = T1 * T2
A = T3

Quadruple:

	Operator	***Operand 1***	***Operand 2***	***Result***
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

Triple:

	Operator	***Operand 1***	***Operand 2***
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	Statement
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	Operator	***Operand 1***	***Operand 2***
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

Translation of Assignment Statements

A statement A := - B * (C + D) has the following three-address translation:

T1 := - B
T2 := C+D
T3 := T1* T2
A := T3

Production	**Semantic Action**
S → id := E	S.code = E.code || gen(id.place = E.place)
E → E1 + E2	E.place = newtemp(); E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)
E → E1 * E2	E.place = newtemp(); E.code = E1.code || E2.code || gen(E.place = E1.place * E2.place)
E → - E1	E.place = newtemp(); E.code = E1.code || gen(E.place = - E1.place)
E → (E1)	E.place = E1.place; E.code = E1.code
E → id	E.place = id.place; E.code = null

Translation of Boolean Expressions

Grammar for Boolean Expressions is:

E → E or E
E → E and E
E → not E
E → (E)
E → id
E → id relop id

There are two representations viz. Numerical and Control-Flow.

Numerical Representation of Boolean

- TRUE is denoted by 1 and FALSE by 0.
- Expressions are evaluated from left to right, in a manner similar to arithmetic expressions.

Example:
The translation for **A or B and C** is the three-address sequence:

T1 := B and C
T2 := A or T1

Also, the translation of a relational expression such as A < B is the three-address sequence:

(1) if A < B goto (4)
(2) T := 0
(3) goto (5)
(4) T := 1
(5)

Therefore, a Boolean expression A < B or C can be translated as:

(1) if A < B goto (4)
(2) T1 := 0
(3) goto (5)
(4) T1 := 1
(5) T2 := T1 or C

Production	**Semantic Action**
E → E1 or E2	T = newtemp (); E.place = T; Gen (T = E1.place or E2.place)
E → E1 and E2	T = newtemp (); E.place = T; Gen (T = E1.place and E2.place)
E → not E1	T = newtemp (); E.place = T; Gen (T = not E1.place)
E → (E1)	E.place = E1.place; E.code = E1.code
E → id	E.place = id.place; E.code = null
E → id1 relop id2	T = newtemp (); E.place = T; Gen (if id1.place relop id2.place goto NEXTQUAD+3) Gen (T = 0) Gen (goto NEXTQUAD+2) Gen (T = 1)

- Quadruples are being generated and NEXTQUAD indicates the next available entry in the quadruple array.

Control-Flow Representation of Boolean Expressions

- If we evaluate Boolean expressions by program position, we may be able to avoid evaluating the entire expressions.
- In A or B, if we determine A to be true, we need not evaluate B and can declare the entire expression to be true.
- In A and B, if we determine A to be false, we need not evaluate B and can declare the entire expression to be false.
- A better code can thus be generated using the above properties.

Example:

The statement **if (A<B || C<D) x = y + z;** can be translated as

(1) if A<B goto (4)
(2) if C<D goto (4)
(3) goto (6)
(4) T = y + z
(5) X = T
(6)

Here (4) is a true exit and (6) is a false exit of the Boolean expressions.

Generating 3-address code for Numerical Representation of Boolean expressions

- Consider a production **E → E1 or E2** that represents the OR Boolean expression. If E1 is true, we know that E is true so we make the location TRUE for E1 be the same as TRUE for E. If E1 is false, then we must evaluate E2, so we make FALSE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.
- Consider a production **E → E1 and E2** that represents the AND Boolean expression. If E1 is false, we know that E is false so we make the location FALSE for E1 be the same as FALSE for E. If E1 is true, then we must evaluate E2, so we make TRUE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.
- Consider the production **E → not E** that represents the NOT Boolean expression. We may simply interchange the TRUE and FALSE exits of E1 to get the TRUE and FALSE exits of E.
- To generate quadruples in the manner suggested above, we use three functions-Makelist, Merge and Backpatch that shall work on the list of quadruples as suggested by their name.
- If we need to proceed to E2 after evaluating E1, we have an efficient way of doing this by modifying our grammar as follows:

  E → E or M E
  E → E and M E
  E → not E
  E → (E)
  E → id
  E → id relop id
  M → ε

- The translation scheme for this grammar would as follows:

Production	**Semantic Action**
E → E1 or M E2	BACKPATCH (E1.FALSE, M.QUAD); E.TRUE = MERGE (E1.TRUE, E2.TRUE); E.FALSE = E2.FALSE;
E → E1 and M E2	BACKPATCH (E1.TRUE, M.QUAD); E.TRUE = E2.TRUE; E.FALSE = MERGE (E1.FALSE, E2.FALSE);
E → not E1	E.TRUE = E1.FALSE; E.FALSE = E1.TRUE;

E → (E1)	E.TRUE = E1.TRUE; E.FALSE = E1.FALSE;
E → id	E.TRUE = MAKELIST (NEXTQUAD); E.FALSE = MAKELIST (NEXTQUAD + 1); GEN (if id.PLACE goto _); GEN (goto _);
E → id1 relop id2	E.TRUE = MAKELIST (NEXTQUAD); E.FALSE = MAKELIST (NEXTQUAD + 1); GEN (if id1.PLACE relop id2.PLACE goto _); GEN (goto _);
M → ε	M.QUAD = NEXTQUAD;

Example:

For the expression P<Q or R<S and T, the parsing steps and corresponding semantic actions are shown below. We assume that NEXTQUAD has an initial value of 100.

Step 1: P<Q gets reduced to E by E → id relop id. The grammatical form is E1 or R<S and T.

We have the following code generated (Makelist).

100: if P<Q goto _
101: goto _

E1 is true if goto of 100 is reached and false if goto of 101 is reached.

Step 2: R<S gets reduced to E by E → id relop id. The grammatical form is E1 or E2 and T.

We have the following code generated (Makelist).

102: if R<S goto _
103: goto _

E2 is true if goto of 102 is reached and false if goto of 103 is reached.

Step 3: T gets reduced to E by E → id. The grammatical form is E1 or E2 and E3.

We have the following code generated (Makelist).

104: if T goto _
105: goto _

E3 is true if goto of 104 is reached and false if goto of 105 is reached.

Step 4: E2 and E3 gets reduced to E by E → E and E. The grammatical form is E1 or E4.

We have no new code generated but changes are made in the already generated code (Backpatch).

```
100: if P<Q goto _
101: goto _
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _
```

E4 is true only if E3.TRUE (goto of 104) is reached. E4 is false if E2.FALSE (goto of 103) or E3.FALSE (goto of 105) is reached (Merge).

Step 5: E1 or E4 gets reduced to E by E → E or E. The grammatical form is E.

We have no new code generated but changes are made in the already generated code (Backpatch).

```
100: if P<Q goto _
101: goto 102
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _
```

E is true only if E1.TRUE (goto of 100) or E2.TRUE (goto of 104) is reached (Merge). E is false if E4.FALSE (goto of 103 or 105) is reached.

Mixed Mode Expressions

- Boolean expressions may in practice contain arithmetic sub expressions e.g. (A+B)>C.
- We can accommodate such sub-expressions by adding the production E → E op E to our grammar.
- We will also add a new field MODE for E. If E has been achieved after reduction using the above (arithmetic) production, we make E.MODE = arith, otherwise make E.MODE = bool.
- If E.MODE = arith, we treat it arithmetically and use E.PLACE. If E.MODE = bool, we treat it as Boolean and use E.FALSE and E.TRUE.

Statements that Alter Flow of Control

- In order to implement goto statements, we need to define a LABEL for a statement. A production can be added for this purpose:

```
      S  → LABEL : S
LABEL  → id
```

- The semantic action attached with this production is to record the LABEL and its value (NEXTQUAD) in the symbol table. It will also Backpatch any previous references to this LABEL with its current value.

- Following grammar can be used to incorporate structured Flow-of-control constructs:

(1) S → if E then S
(2) S → if E then S else S
(3) S → while E do S
(4) S → begin L end
(5) S → A

(6) L → L ; S
(7) L → S

Here, S denotes a statement, L a statement-list, A an assignment statement and E a Boolean-valued expression.

Translation Scheme for statements that alter flow of control

- We introduce a new field NEXT for S and L like TRUE and FALSE for E. S.NEXT and L.NEXT are respectively the pointers to a list of all conditional and unconditional jumps to the quadruple following statement S and statement-list L in execution order.
- We also introduce the marker non-terminal M as in the case of grammar for Boolean expressions. This is put before statement in if-then, before both statements in if-then-else and the statement in while-do as we may need to proceed to them after evaluating E. In case of while-do, we also need to put M before E as we may need to come back to it after executing S.
- In case of if-then-else, if we evaluate E to be true, first S will be executed. After this we should ensure that instead of second S, the code after this if-then-else statement be executed. We thus place another non-terminal marker N after first S i.e. before else.
- The grammar now is as follows:

  (1) S → if E then M S
  (2) S → if E then M S N else M S
  (3) S → while M E do M S
  (4) S → begin L end
  (5) S → A
  (6) L → L ; M S
  (7) L → S
  (8) M → ε
  (9) N → ε

- The translation scheme for this grammar would as follows:

Production	**Semantic Action**
S → if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S → if E then M1 S1 N else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S → while M1 E do M2 S1	BACKPATCH (S1.NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S → begin L end	S.NEXT = L.NEXT
S → A	S.NEXT = MAKELIST ( )
L → L1 ; M S	BACKPATCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L → S	L.NEXT = S.NEXT

M → ε	M.QUAD = NEXTQUAD
N → ε	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto _)

Postfix Translations

- In an production A → α, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α.
- Productions can be factored to achieve Postfix form.

Postfix translation of while statement

The production

S → while M1 E do M2 S1

can be factored as

S → C S1
C → W E do
W → while

A suitable translation scheme would be

Production	Semantic Action
W → while	W.QUAD = NEXTQUAD
C → W E do	C.QUAD = W.QUAD BACKPATCH (E.TRUE, NEXTQUAD) C.FALSE = E.FALSE
S → C S1	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Postfix translation of for statement

Consider the following production which stands for the for-statement

S → for L = E1 step E2 to E3 do S1

Here L is any expression with l-value, usually a variable, called the index. E1, E2 and E3 are expressions called the initial value, increment and limit, respectively. Semantically, the for-statement is equivalent to the following program.

```
begin
        INDEX = addr ( L );
        *INDEX = E1;
        INCR = E2;
        LIMIT = E3;
        while *INDEX <= LIMIT do
            begin
```

```
                    code for statement S1;
                    *INDEX = *INDEX + INCR;
          end
end
```

The non-terminals L, E1, E2, E3 and S appear in the same order as in the production. The production can be factored as

(1) F → for L
(2) T → F = E1 by E2 to E3 do
(3) S → T S1

A suitable translation scheme would be

Production	**Semantic Action**
F → for L	F.INDEX = L.INDEX
T → F = E1 by E2 to E3 do	GEN (*F.INDEX = E1.PLACE) INCR = NEWTEMP ( ) LIMIT = NEWTEMP ( ) GEN (INCR = E2.PLACE) GEN (LIMIT = E3.PLACE) T.QUAD = NEXTQUAD T.NEXT = MAKELIST (NEXTQUAD) GEN (IF *F.INDEX > LIMIT goto _) T.INDEX = F.INDEX T.INCR = INCR
S → T S1	BACKPATCH (S1.NEXT, NEXTQUAD) GEN (*T.INDEX = *T.INDEX + T.INCR) GEN (goto T.QUAD) S.NEXT = T.NEXT

Translation with a Top-Down Parser

- Any translation done by top-down parser can be done in a bottom-up parser also.
- But in certain situations, translation with a top-down parser is advantageous as tricks such as placing a marker non-terminal can be avoided.
- Semantic routines can be called in the middle of productions in top-down parser.

Array references in arithmetic expressions

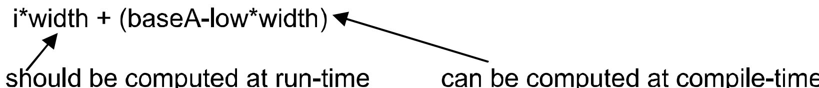
- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.
- For a one-dimensional array A:

Base (A) is the address of the first location of the array A,
width is the width of each array element.
low is the index of the first array element

location of A[i] = baseA+(i-low)*width

baseA+(i-low)*width

can be re-written as



- So, the location of A[i] can be computed at the run-time by evaluating the formula i*width+c where c is (baseA-low*width) which is evaluated at compile-time.
- Intermediate code generator should produce the code to evaluate this formula i*width+c (one multiplication and one addition operation).
- A two-dimensional array can be stored in either row-major (row-by-row) or column-major (column-by-column).
- Most of the programming languages use row-major method.
- The location of A[i1,i2] is baseA+ ((i1-low1)*n2+i2-low2)*width

baseA is the location of the array A.
low1 is the index of the first row
low2 is the index of the first column
n2 is the number of elements in each row
width is the width of each array element

Again, this formula can be re-written as

((i1*n2)+i2)*width + (baseA-((low1*n1)+low2)*width)

should be computed at run-time   can be computed at compile-time

- Arrays of any dimension can be dealt in a similar but general manner.

In general, the location of A[i1,i2,...,ik] is

((... ((i1*n2)+i2) ...)*nk+ik)*width + (baseA-((...((low1*n1)+low2)...)*nk+lowk)*width)

So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of A[i1,i2,...,ik]) :

((... ((i1*n2)+i2) ...)*nk+ik)*width + c

To evaluate the ((... ((i1*n2)+i2) ...)*nk+ik portion of this formula, we can use the recurrence equation:

e1 = i1
em = em-1 * nm + im

Grammar and Translation Scheme

The grammar and suitable translation scheme for arithmetic expressions with array references is as given below:

<u>**Production**</u>	<u>**Semantic Action**</u>
S → L = E	if (L.OFFSET = NULL) then GEN (L.PLACE = E.PLACE) else GEN(L.PLACE [L.OFFSET] = E.PLACE)
E → E1 + E2	E.PLACE = NEWTEMP ( )

	GEN (E.PLACE = E1.PLACE + E2.PLACE)
E → (E1)	E.PLACE = E1.PLACE
E → L	if (L.OFFSET = NULL) then E.PLACE = L.PLACE else {E.PLACE = NEWTEMP ( ); GEN (E.PLACE = L.PLACE[L.OFFSET])}
L → id	L.PLACE = id.PLACE L.OFFSET = NULL
L → ELIST]	L.PLACE = NEWTEMP( ) L.OFFSET = NEWTEMP ( ) GEN (L.PLACE = ELIST.ARRAY - C) GEN (L.OFFSET = ELIST.PLACE * WIDTH (ELIST.ARRAY))
ELIST → ELIST1 , E	ELIST.ARRAY = ELIST1.ARRAY ELIST.PLACE = NEWTEMP ( ) ELIST.NDIM = ELIST1.NDIM + 1 GEN (ELIST.PLACE = ELIST1.PLACE * LIMIT (ELIST.ARRAY, ELIST.NDIM)) GEN (ELIST.PLACE = E.PLACE + ELIST.PLACE)
ELIST → id [E	ELIST.ARRAY = id.PLACE ELIST.PLACE = E.PLACE ELIST.NDIM = 1

Here, NDIM denotes the number of dimensions, LIMIT (AARAY, i) function returns the upper limit along the ith dimension of ARRAY i.e. ni, WIDTH (ARRAY) returns the number of bytes for one element of ARRAY.

Declarations

Following is the grammar and a suitable translation scheme for declaration statements:

Production	Semantic Action
D → integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D → real, id	ENTER (id.PLACE, real) D.ATTR = real
D → D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

Here, ENTER makes the entry into symbol table while ATTR is used to trace the data type.

Procedure Calls

Following is the grammar and a suitable translation scheme for Procedure Calls:

Production	Semantic Action
S → call id (ELIST)	for each item p on QUEUE do GEN (param p) GEN (call id.PLACE)

ELIST → ELIST, E   append E.PLACE to the end of QUEUE

ELIST → E   initialize QUEUE to contain only E.PLACE

QUEUE is used to store the list of parameters in the procedure call.

Case Statements

The case statement has following syntax:

```
switch E
        begin
                case V1: S1
                case V2: S2
                .
                .
                .
                case Vn-1: Sn-1
                default: Sn
        end
```

The translation scheme for this shown below:

```
                code to evaluate E into T
                goto TEST
        L1:     code for S1
                goto NEXT
        L2:     code for S2
                goto NEXT
                .
                .
                .
        Ln-1:   code for Sn-1
                goto NEXT
        Ln:     code for Sn
                goto NEXT
        TEST:   if T = V1 goto L1
                If T = V2 goto L2
                .
                .
                .
                if T = Vn-1 goto Ln-1
                goto Ln
NEXT:
```

MODULE-3 TYPE CHECKING

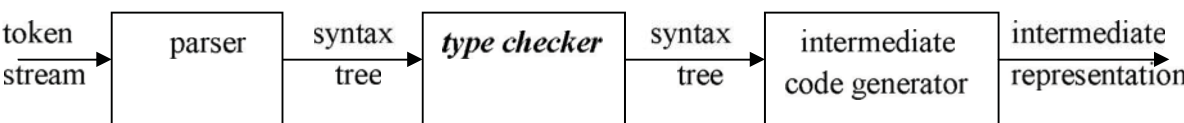
TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking,* detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks –** Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.
   Constructors include:
   Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

   Products : If T_1 and T_2 are type expressions, then their Cartesian product T_1 X T_2 is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.

For example:

type row = record
address: integer;
lexeme: array[1..15] of char
end;
var table: array[1...101] of row;

declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

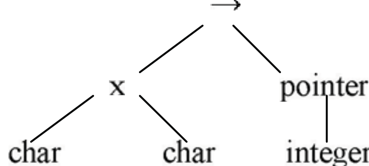
Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".

For example, ***var p: ↑ row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → ***pointer*** **(integer)**



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic.
- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

Sound type system

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E	
D → D ; D	
D → id : T	{ *addtype* (id.*entry* , T.*type*) }
T → char	{ T.*type* : = char }
T → integer	{ T.*type* : = integer }
T → ↑ T1	{ T.*type* : = pointer(T_1.*type*) }
T → array [num] of T1	{ T.*type* : = array (1… num.val , T_1.*type*) }

In the above language,
→ There are two basic types : char and integer ;
→ *type_error* is used to signal errors;
→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* for E gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E_1 **mod** E_2 { *E.type* : = **if** E_1. *type* = *integer* **and**
   E_2. *type* = *integer* **then** *integer*
   else *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E_1 [E_2] { *E.type* : = if E_2.*type* = *integer* **and**
   E_1.*type* = *array(s,t)* **then** *t*
   else *type_error* }
   In an array reference E_1 [E_2] , the index expression E_2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

5. E → E_1 ↑ { *E.type* : = **if** E_1.*type* = *pointer (t)* **then** *t*
   else *type_error* }

   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void*
else *type_error* }

2. Conditional statement:

S → **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

$S \rightarrow S_1 \; ; \; S_2$ { $S.type := \textbf{if } S_1.type = void$ and
$S_1.type = void \textbf{ then } void$
else *type_error* }

Type checking of functions

The rule for checking the type of a function application is :

$E \rightarrow E_1 (E_2)$ { $E.type := \textbf{if } E_2.type = s \textbf{ and}$
$E_1.type = s \rightarrow t \textbf{ then } t$
else *type_error* }

MODULE 4 - RUN-TIME ENVIRONMENTS

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. The identifier is the *procedure name*, and the statement is the *procedure body*.

For example, the following is the definition of procedure named *readarray* :

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.

- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.
Declarations may be explicit, such as:

var i : integer ;

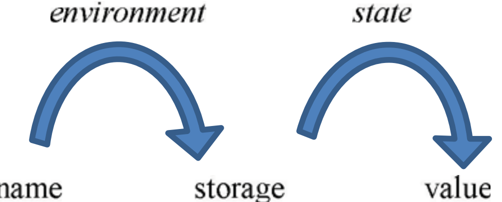
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

Binding of names:

Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values.

The term *environment* refers to a function that maps a name to a storage location.
The term *state* refers to a function that maps a storage location to the value held there.

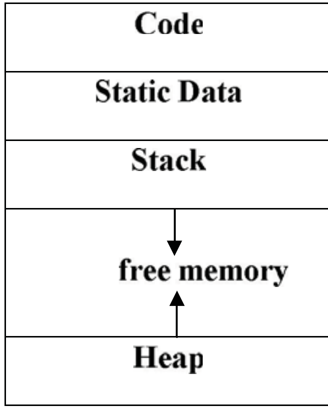


When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

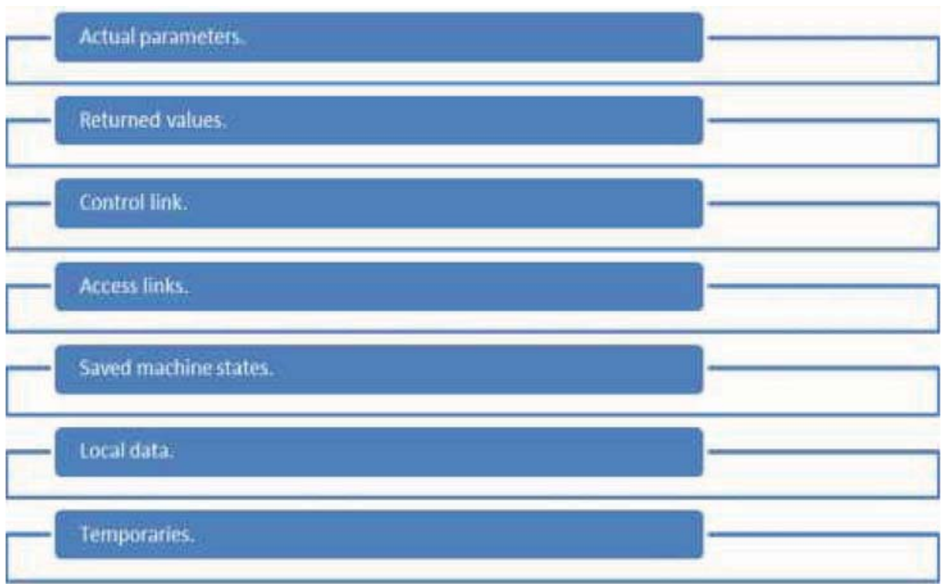
Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

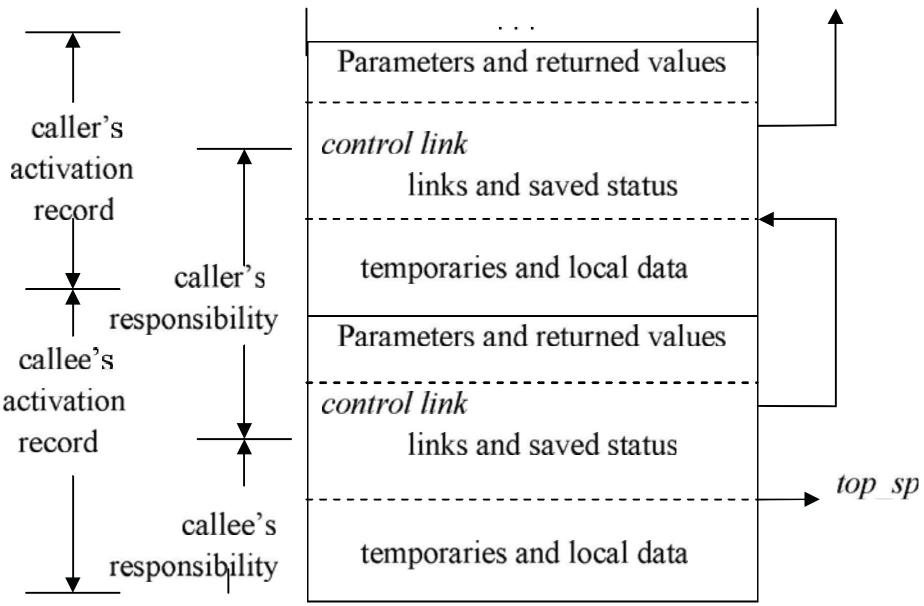
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.

- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

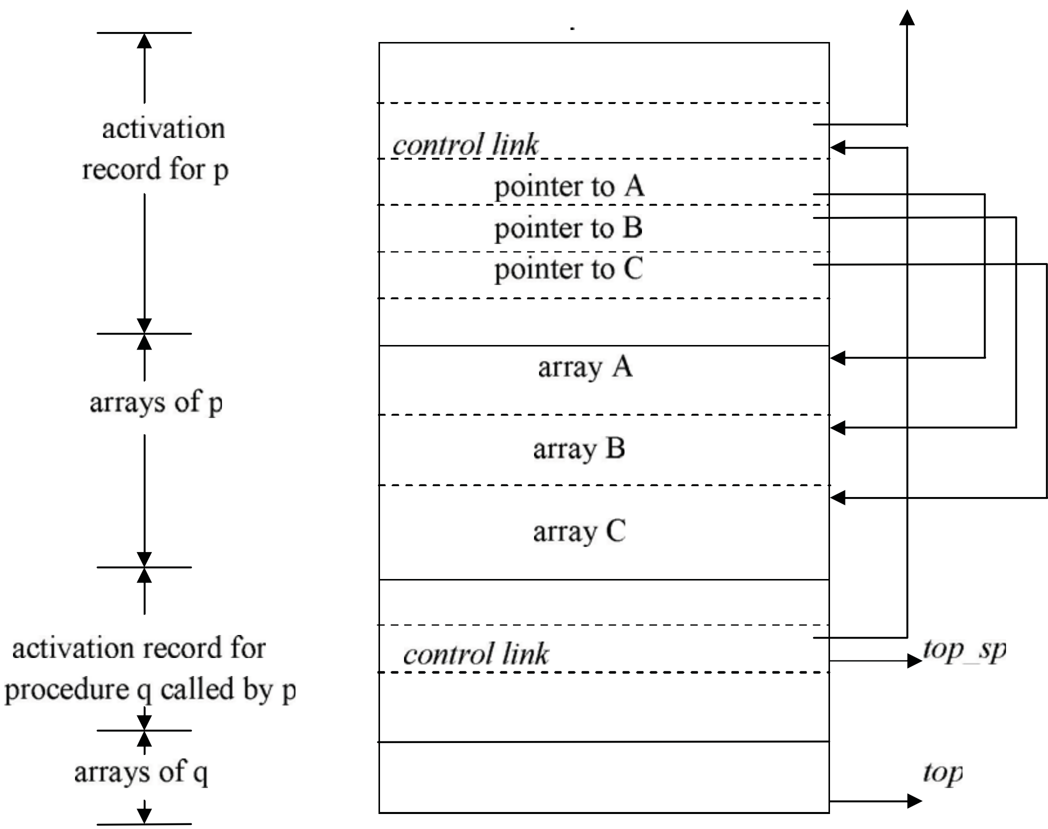


Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.
- A suitable, corresponding return sequence is:
  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
s r q (1 , 9)	s *control link* r *control link* q(1,9) *control link*	Retained activation record for r

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

MODULE-4 INTERMEDIATE CODE GENERATION

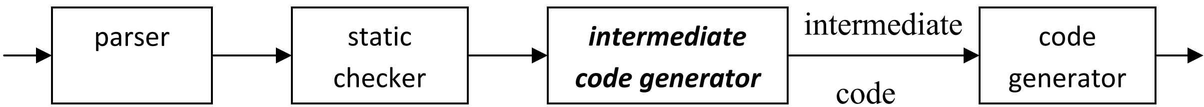
INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

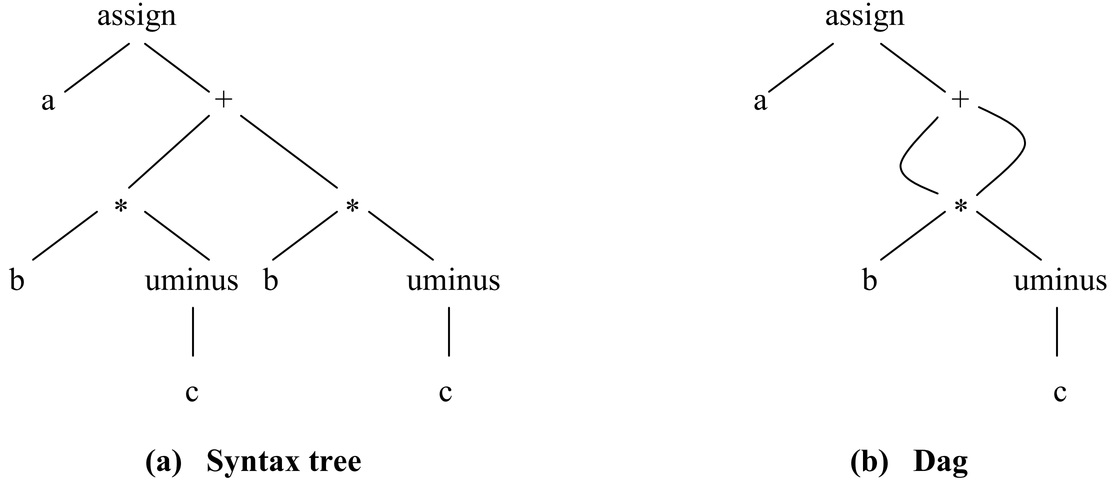
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A **dag (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement **a : = b * - c + b * - c** are as follows:



(a) Syntax tree **(b) Dag**

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

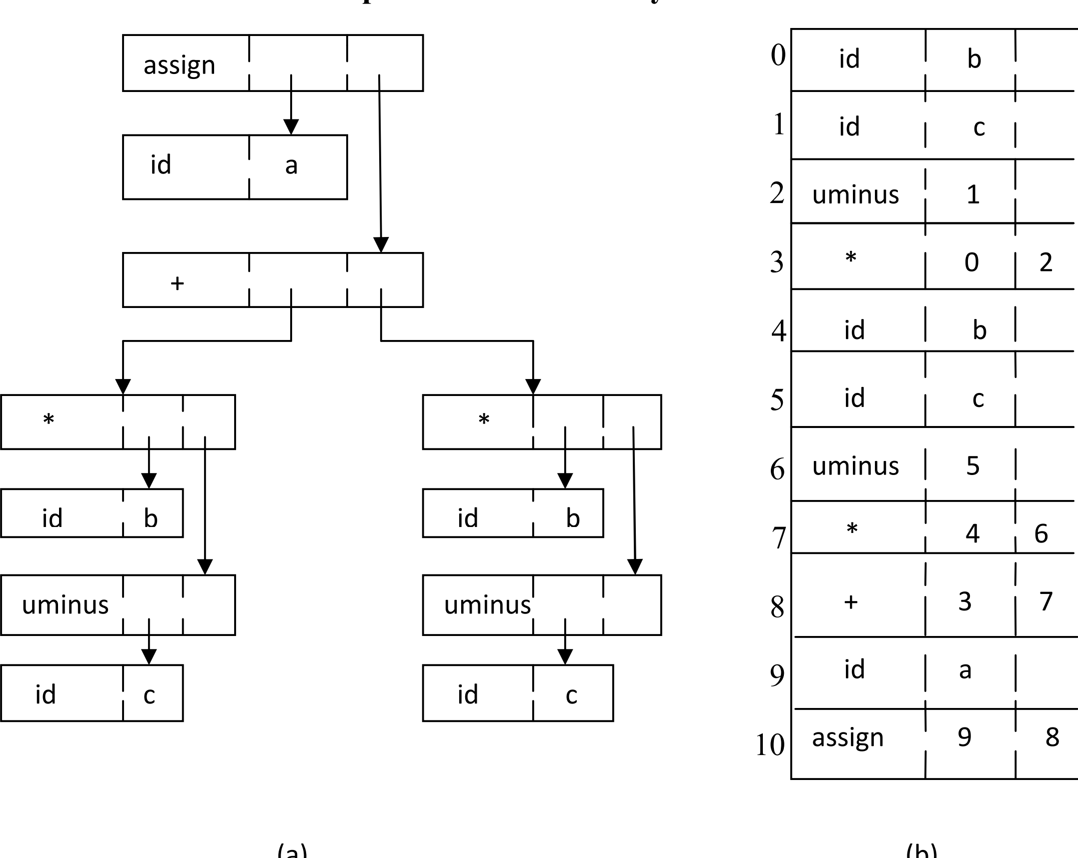
PRODUCTION	SEMANTIC RULE
S → id : = E	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
E → E_1 + E_2	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
E → E_1 * E_2	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
E → - E_1	E.nptr : = mknode('uminus', E_1.nptr)
E → (E_1)	E.nptr : = E_1.nptr
E → id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id.** If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



(a) (b)

Three-Address Code:

Three-address code is a sequence of statements of the general form

x : = y *op* z

where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

t_1 : = y * z
t_2 : = x + t_1

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

(a) Code for the syntax tree	(b) Code for the dag
$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x : = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x : = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x : = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation

relop to y. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.

6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,

    param x_1
    param x_2
    . . .
    param x_n
    call p,n

    generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example, **id : = *E*** consists of code to evaluate ***E*** into some temporary t, followed by the assignment **id**.*place* : = **t.**

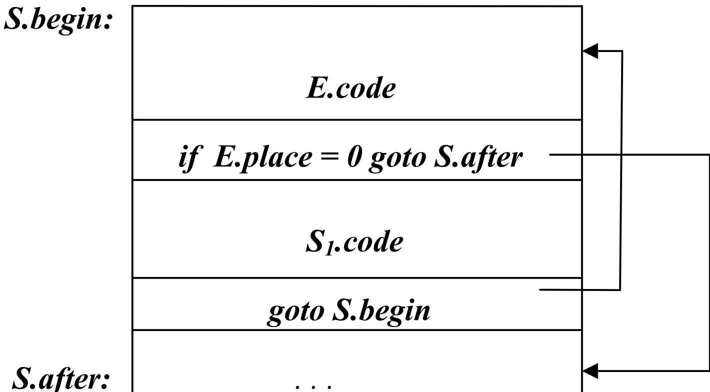
Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute *S.code* represents the three-address code for the assignment *S.*
The nonterminal *E* has two attributes :

1. *E.place*, the name that will hold the value of *E* , and
2. *E.code*, the sequence of three-address statements evaluating *E*.

Syntax-directed definition to produce three-address code for assignments

PRODUCTION	SEMANTIC RULES
S → id : = E	***S.code : = E.code || gen(id.place ':=' E.place)***
E → E₁ + E₂	***E.place := newtemp;*** ***E.code := E₁.code || E₂.code || gen(E.place ':=' E₁.place '+' E₂.place)***
E → E₁ * E₂	***E.place := newtemp;*** ***E.code := E₁.code || E₂.code || gen(E.place ':=' E₁.place '*' E₂.place)***
E → - E₁	***E.place := newtemp;*** ***E.code := E₁.code || gen(E.place ':=' 'uminus' E₁.place)***
E → (E₁)	***E.place : = E₁.place;*** ***E.code : = E₁.code***
E → id	***E.place : = id.place;*** ***E.code : = ' '***

Semantic rules generating code for a while statement



PRODUCTION	**SEMANTIC RULES**
S → **while** *E* **do** S_1	***S.begin := newlabel;*** ***S.after := newlabel;*** ***S.code := gen(S.begin ':') ||*** ***E.code ||*** ***gen ('if' E.place '=' '0' 'goto' S.after)||*** ***S_1.code ||*** ***gen ('goto' S.begin) ||*** ***gen (S.after ':')***

- The function *newtemp* returns a sequence of distinct names t_1,t_2,….. in response to successive calls.
- Notation *gen(x ':=' y '+' z)* is used to represent three-address statement x := y + z. Expressions appearing instead of variables like *x, y* and *z* are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for ***S*** **→ while *E* do *S₁*** is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for *E* and the statement following the code for S, respectively.
- The function *newlabel* returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of ***E*** becomes zero, control leaves the while statement.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement **x : = y op z** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1* and *arg2.*
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] : = y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(b) x : = y[i]

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form **id :** ***T.***
- Before the first declaration is considered, *offset* is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.
- The procedure *enter(name, type, offset*) creates a symbol-table entry for *name*, gives its type *type* and relative address *offset* in its data area.
- Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors *pointer* and *array*. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

P → D	***{ offset : = 0 }***
D → D ; D	
D → id : T	***{ enter(id.name, T.type, offset); offset : = offset + T.width }***
T → integer	***{ T.type : = integer; T.width : = 4 }***
T → real	***{ T.type : = real; T.width : = 8 }***
T → array [num] of T_1	***{ T.type : = array(num.val,*** T_1***.type); T.width : = num.val X*** T_1***.width }***
T → ↑ T_1	***{ T.type : = pointer (*** T_1***.type); T.width : = 4 }***

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

$P \rightarrow D$

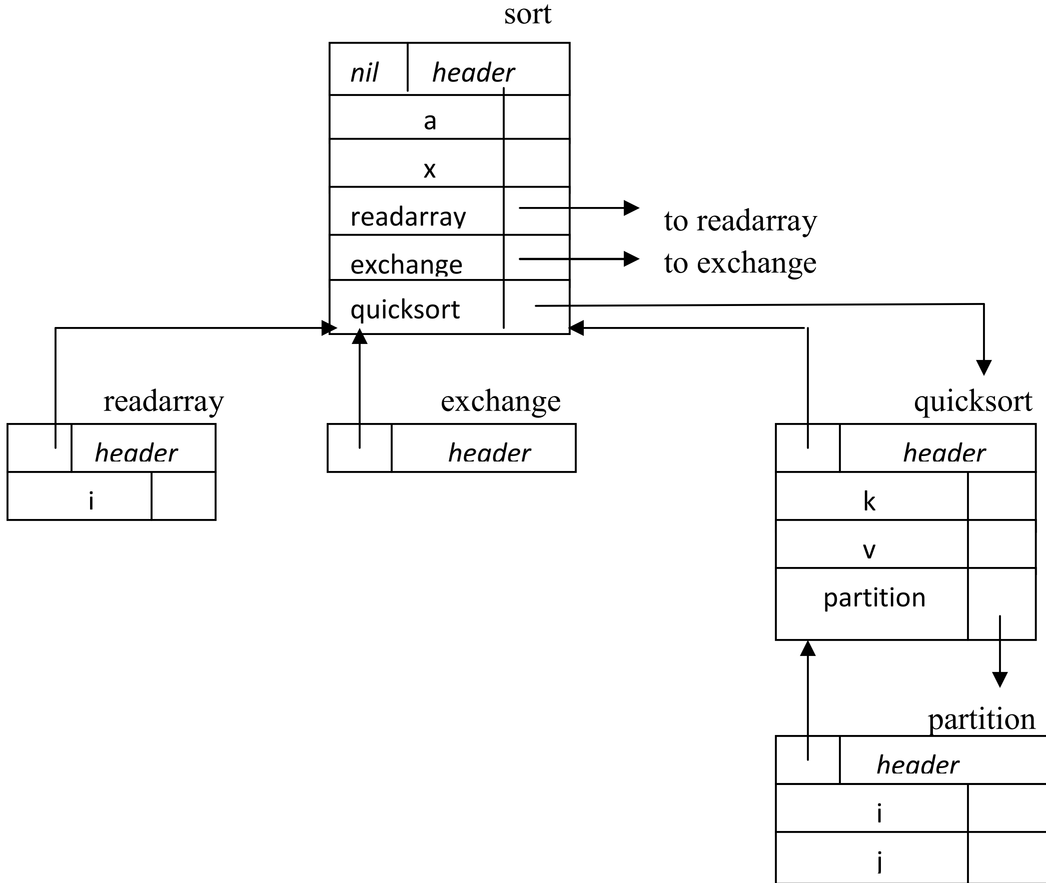
$D \rightarrow D\ ;\ D$ | **id** : T | **proc id** ; D ; S

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration $D \rightarrow$ ***proc id*** $D_1;S$ is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure *enter* is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures *readarray, exchange*, and *quicksort* pointing back to that for the containing procedure *sort*, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

1. *mktable(previous)* creates a new symbol table and returns a pointer to the new table. The argument *previous* points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)* creates a new entry for name *name* in the symbol table pointed to by *table.* Again, *enter* places type *type* and relative address *offset* in fields within the entry.

3. *addwidth(table, width)* records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Syntax directed translation scheme for nested procedures

P → M D	***{ addwidth (top(tblptr) , top (offset)); pop (tblptr); pop (offset) }***
M → ε	***{ t : = mktable (nil); push (t,tblptr); push (0,offset) }***
D → D_1 ; D_2	
D → proc id ; N D_1 ; S	***{ t : = top (tblptr); addwidth (t, top (offset)); pop (tblptr); pop (offset); enterproc (top (tblptr), id.name, t) }***
D → id : T	***{ enter (top (tblptr), id.name, T.type, top (offset)); top (offset) := top (offset) + T.width }***
N → ε	***{ t := mktable (top (tblptr)); push (t, tblptr); push (0,offset) }***

- The stack *tblptr* is used to contain pointers to the tables for **sort, quicksort,** and **partition** when the declarations in **partition** are considered.

- The top element of stack *offset* is the next available relative address for a local of the current procedure.

- All semantic actions in the subtrees for B and C in

  A → BC {$action_A$}

  are done before $action_A$ at the end of the production occurs. Hence, the action associated with the marker M is the first to be done.

- The action for nonterminal M initializes stack *tblptr* with a symbol table for the outermost scope, created by operation *mktable(nil).* The action also pushes relative address 0 onto stack offset.

- Similarly, the nonterminal N uses the operation *mktable(top(tblptr))* to create a new symbol table. The argument *top(tblptr)* gives the enclosing scope for the new table.

- For each variable declaration **id:** T, an entry is created for **id** in the current symbol table. The top of stack offset is incremented by T.width.

- When the action on the right side of *D → proc id; ND₁; S* occurs, the width of all declarations generated by D_1 is on the top of stack offset; it is recorded using *addwidth.* Stacks *tblptr* and *offset* are then popped.
  At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar.

P → M D

M → ε

D → D ; D | **id** : T | **proc id** ; N D ; S

N → ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

```
S → id : = E        { p : = lookup ( id.name);
                      if  p ≠ nil then
                      emit( p ‘ : =’ E.place)
                      else error }

E → E1 + E2         { E.place : = newtemp;
                      emit( E.place ‘: =’ E1.place ‘ + ‘ E2.place ) }

E → E1 * E2         { E.place : = newtemp;
                      emit( E.place ‘: =’ E1.place ‘ * ‘ E2.place ) }

E → - E1            { E.place : = newtemp;
                      emit ( E.place ‘: =’  ‘uminus’ E1.place ) }

E → ( E1 )          { E.place : = E1.place }
```

```
E → id            { p : = lookup ( id.name);
                     if p ≠ nil then
                         E.place : = p
                     else error }
```

Reusing Temporary Names

- The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.
- Temporaries can be reused by changing *newtemp*. The code generated by the rules for $E \rightarrow E_1 + E_2$ has the general form:

  evaluate E_1 into t_1
  evaluate E_2 into t_2
  $t := t_1 + t_2$

- The lifetimes of these temporaries are nested like matching pairs of balanced parentheses.
- Keep a count c , initialized to zero. Whenever a temporary name is used as an operand, decrement c by 1. Whenever a new temporary name is generated, use $c and increase c by 1.
- For example, consider the assignment x := a * b + c * d – e * f

Three-address code with stack temporaries

statement	*value of c*
	0
$0 := a * b	1
$1 := c * d	2
$0 := $0 + $1	1
$1 := e * f	2
$0 := $0 - $1	1
x := $0	0

Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *w*, then the *i*th element of array A begins in location

$$base + (i - low) \times w$$

where *low* is the lower bound on the subscript and *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of A*[low]*.

The expression can be partially evaluated at compile time if it is rewritten as

$$i \times w + (base - low \times w)$$

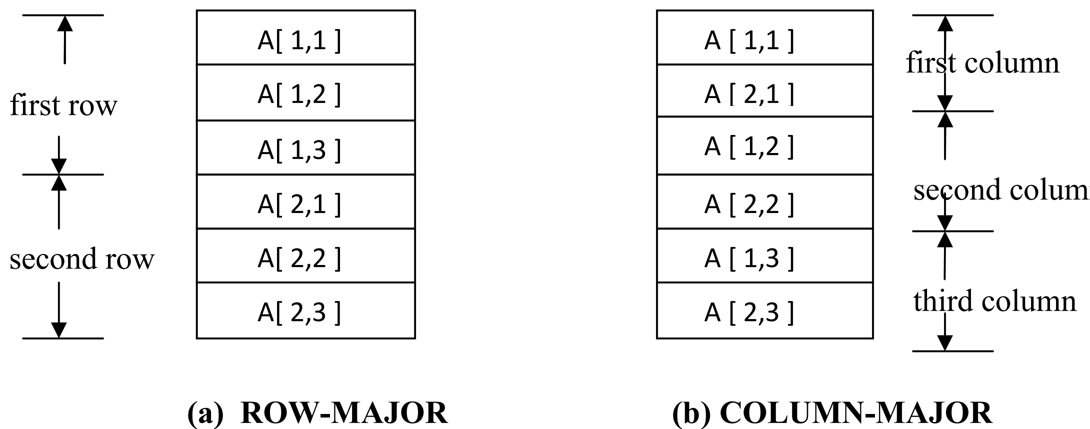
The subexpression $c = base - low \times w$ can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of A[i] is obtained by simply adding $i \times w$ to c.

Address calculation of multi-dimensional arrays:

A two-dimensional array is stored in of the two forms :

- Row-major (row-by-row)
- Column-major (column-by-column)

Layouts for a 2 x 3 array



(a) ROW-MAJOR **(b) COLUMN-MAJOR**

In the case of row-major form, the relative address of A[i_1 , i_2] can be calculated by the formula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

where, low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if $high_2$ is the upper bound on the value of i_2, then $n_2 = high_2 - low_2 + 1$.

Assuming that i_1 and i_2 are the only values that are known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

Generalized formula:

The expression generalizes to the following expression for the relative address of A[$i_1, i_2, \ldots, i_k$]

$$((\ldots ((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w + base - ((\ldots ((low_1 n_2 + low_2) n_3 + low_3) \ldots) n_k + low_k) \times w$$

for all j, $n_j = high_j - low_j + 1$

The Translation Scheme for Addressing Array Elements :

Semantic actions will be added to the grammar :

(1) $S \rightarrow L := E$
(2) $E \rightarrow E + E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow L$
(5) $L \rightarrow Elist$]
(6) $L \rightarrow$ **id**
(7) $Elist \rightarrow Elist , E$
(8) $Elist \rightarrow$ **id** [E

We generate a normal assignment if L is a simple name, and an indexed assignment into the location denoted by L otherwise :

(1) $S \rightarrow L := E$ { **if** *L.offset* = **null then** / * *L is a simple* **id** */
emit (*L.place* ': =' *E.place*) ;
else
emit (*L.place* ' [' *L.offset* ']' ': =' *E.place*) }

(2) $E \rightarrow E_1 + E_2$ { *E.place* : = *newtemp;*
emit (*E.place* ': =' E_1*.place* ' + ' E_2*.place*) }

(3) $E \rightarrow (E_1)$ { *E.place* : = E_1*.place* }

When an array reference L is reduced to E , we want the *r*-value of L. Therefore we use indexing to obtain the contents of the location *L.place* [*L.offset*] :

(4) $E \rightarrow L$ { **if** *L.offset* = **null then** /* *L is a simple* **id** * /
E.place : = *L.place*
else begin
E.place : = *newtemp;*
emit (*E.place* ': =' *L.place* ' [' *L.offset* ']')
end }

(5) $L \rightarrow Elist$]   { *L.place* : = *newtemp;*
L.offset : = *newtemp;*
emit (*L.place* ': =' *c(Elist.array));*
emit (*L.offset* ': =' *Elist.place* '*' *width (Elist.array)*) }

(6) $L \rightarrow$ **id**   { *L.place* := **id**.*place;*
L.offset := **null** }

(7) $Elist \rightarrow Elist_1 , E$ { *t* := *newtemp;*
m : = $Elist_1$*.ndim* + 1*;*
emit (*t* ': =' $Elist_1$*.place* '*' *limit* ($Elist_1$*.array,m*))*;*
emit (*t* ': =' *t* '+' *E.place*)*;*
Elist.array : = $Elist_1$*.array;*

Elist.place : = t;
Elist.ndim : = m }

(8) *Elist* → **id** [*E* { *Elist.array : =* **id**.*place;*

Elist.place : = E.place;
Elist.ndim : = 1 }

Type conversion within Assignments :

Consider the grammar for assignment statements as above, but suppose there are two types – real and integer , with integers converted to reals when necessary. We have another attribute *E.type*, whose value is either *real* or *integer*. The semantic rule for *E.type* associated with the production *E* → *E* + *E* is :

E → *E* + *E* { *E.type* : =
if E_1*.type = integer* **and**
E_2*.type = integer* **then** *integer*
else *real* }

The entire semantic rule for *E* → *E* + *E* and most of the other productions must be modified to generate, when necessary, three-address statements of the form x : = inttoreal y, whose effect is to convert integer y to a real of equal value, called x.

Semantic action for *E* → E_1 + E_2

E.place := newtemp;
if E_1*.type = integer* **and** E_2*.type = integer* **then begin**
emit(E.place ': =*' E_1.place '*int +*' E_2.place);*
E.type : = integer
end
else if E_1*.type = real* **and** E_2*.type = real* **then begin**
emit(E.place ': =*' E_1.place* 'real +' E_2*.place);*
E.type : = real
end
else if E_1*.type = integer* **and** E_2*.type = real* **then begin**
u : = newtemp;
emit(u ': =*'* 'inttoreal' E_1*.place);*
emit(E.place ': =*' u* ' real +' E_2*.place);*
E.type : = real
end
else if E_1*.type = real* **and** E_2*.type =integer* **then begin**
u : = newtemp;
emit(u ': =*'* 'inttoreal' E_2*.place*);
emit(E.place ': =*' E_1.place* ' real +' *u);*
E.type : = real
end
else
E.type : = type_error;

For example, for the input x : = y + i * j
assuming *x* and *y* have type *real*, and i and j have type *integer*, the output would look like

t_1 : = i int* j
t_3 : = inttoreal t_1
t_2 : = y real+ t_3
x : = t_2

BOOLEAN EXPRESSIONS

Boolean expressions have two primary purposes. They are used to compute logical values, but more often they are used as conditional expressions in statements that alter the flow of control, such as if-then-else, or while-do statements.

Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **relop** E_2, where E_1 and E_2 are arithmetic expressions.

Here we consider boolean expressions generated by the following grammar :

E → E **or** E | E **and** E | **not** E | (E) | **id relop id | true | false**

Methods of Translating Boolean Expressions:

There are two principal methods of representing the value of a boolean expression. They are :

- To encode true and false ***numerically*** and to evaluate a boolean expression analogously to an arithmetic expression. Often, 1 is used to denote true and 0 to denote false.
- To implement boolean expressions by ***flow of control***, that is, representing the value of a boolean expression by a position reached in a program. This method is particularly convenient in implementing the boolean expressions in flow-of-control statements, such as the if-then and while-do statements.

Numerical Representation

Here, 1 denotes true and 0 denotes false. Expressions will be evaluated completely from left to right, in a manner similar to arithmetic expressions.

For example :

- The translation for
  a **or** b **and** **not** c
  is the three-address sequence
  t_1 : = **not** c
  t_2 : = b **and** t_1
  t_3 : = a **or** t_2
- A relational expression such as a < b is equivalent to the conditional statement
  if a < b then 1 else 0

which can be translated into the three-address code sequence (again, we arbitrarily start statement numbers at 100) :

```
100 :   if a < b goto 103
101 :   t : = 0
102 :   goto 104
103 :   t : = 1
104 :
```

Translation scheme using a numerical representation for booleans

$E \rightarrow E_1$ **or** E_2	{ *E.place : = newtemp;* *emit(E.place ': =' E_1.place* **'or'** *E_2.place) }*
$E \rightarrow E_1$ **and** E_2	{ *E.place : = newtemp;* *emit(E.place ': =' E_1.place* **'and'** *E_2.place) }*
$E \rightarrow$ **not** E_1	{ *E.place : = newtemp;* *emit(E.place ': ='* **'not'** *E_1.place) }*
$E \rightarrow (E_1)$	{ *E.place : = E_1.place* }

$E \rightarrow$ **id_1 relop id_2**	{ *E.place : = newtemp;*
	 emit('if' **id_1**.*place* **relop**.*op* **id_2**.*place* **'goto'** *nextstat* + **3**);
	 emit(E.place ': =' **'0'**);
	 *emit(***'goto'** *nextstat* +**2**);
	 emit(E.place ': =' **'1'**) }

$E \rightarrow$ **true**	{ *E.place : = newtemp;* *emit(E.place ': ='* **'1'**) }
$E \rightarrow$ **false**	{ *E.place : = newtemp;* *emit(E.place ': ='* **'0'**) }

Short-Circuit Code:

We can also translate a boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called "**short-circuit**" or "**jumping**" code. It is possible to evaluate boolean expressions without generating code for the boolean operators **and, or,** and **not** if we represent the value of an expression by a position in the code sequence.

Translation of a < b or c < d and e < f

100 : if a < b goto 103	107 : t_2 : = 1
101 : t_1 : = 0	108 : if e < f goto 111
102 : goto 104	109 : t_3 : = 0
103 : t_1 : = 1	110 : goto 112
104 : if c < d goto 107	111 : t_3 : = 1
105 : t_2 : = 0	112 : t_4 : = t_2 and t_3
106 : goto 108	113 : t_5 : = t_1 or t_4

Flow-of-Control Statements

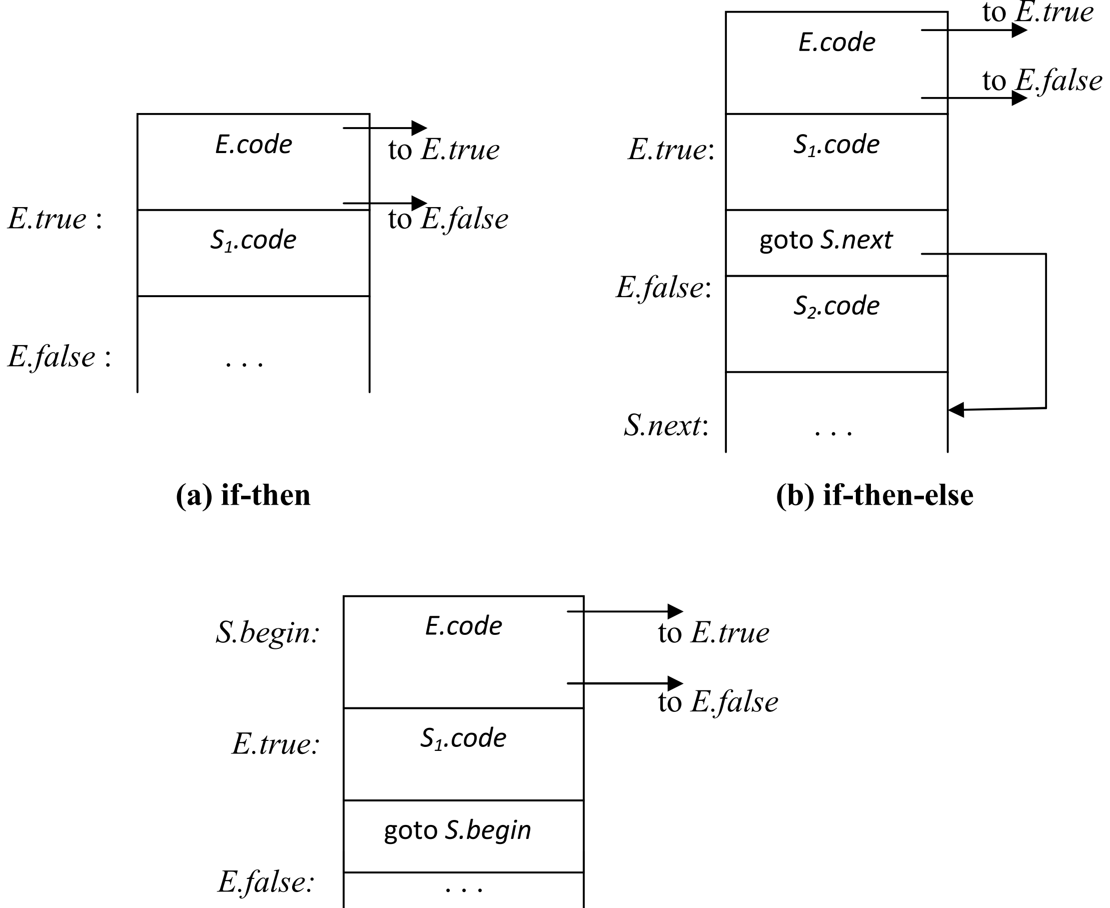
We now consider the translation of boolean expressions into three-address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar:

S → **if** E **then** S_1
  | **if** E **then** S_1 **else** S_2
  | **while** E **do** S_1

In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.next is a label that is attached to the first three-address instruction to be executed after the code for S.

Code for if-then , if-then-else, and while-do statements



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel;$ $E.false := newlabel;$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code \parallel$ $gen('\textbf{goto}'\ S.next) \parallel$ $gen(E.false\ ':') \parallel S_2.code$
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $E.true := newlabel;$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := gen(S.begin\ ':') \parallel E.code \parallel$ $gen(E.true\ ':') \parallel S_1.code \parallel$ $gen('\textbf{goto}'\ S.begin)$

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.false\ ':') \parallel E_2.code$
$E \rightarrow E_1$ **and** E_2	$E.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \parallel gen(E_1.true\ ':') \parallel E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E1)$	$E_1.true := E.true;$

	E_1.*false* : = *E.false;* *E.code* : = E_1.*code*
E → $\mathbf{id_1}$ **relop** $\mathbf{id_2}$	*E.code* : = *gen*('**if**' $\mathbf{id_1}$.*place* **relop.***op* $\mathbf{id_2}$.*place* '**goto**' *E.true*) || *gen*('**goto**' *E.false*)
E → **true**	*E.code* : = *gen*('**goto**' *E.true*)
E → **false**	*E.code* : = *gen*('**goto**' *E.false*)

CASE STATEMENTS

The "switch" or "case" statement is available in a variety of languages. The switch-statement syntax is as shown below :

Switch-statement syntax

switch *expression*
  begin
    case *value* : *statement*
    case *value* : *statement*
      . . .
    case *value* : *statement*
    default : *statement*
  end

There is a selector expression, which is to be evaluated, followed by *n* constant values that the expression might take, including a default "value" which always matches the expression if no other value does. The intended translation of a switch is code to:

1. Evaluate the expression.
2. Find which value in the list of cases is the same as the value of the expression.
3. Execute the statement associated with the value found.

Step (2) can be implemented in one of several ways :

- By a sequence of conditional **goto** statements, if the number of cases is small.
- By creating a table of pairs, with each pair consisting of a value and a label for the code of the corresponding statement. Compiler generates a loop to compare the value of the expression with each value in the table. If no match is found, the default (last) entry is sure to match.
- If the number of cases s large, it is efficient to construct a hash table.
- There is a common special case in which an efficient implementation of the n-way branch exists. If the values all lie in some small range, say i_{min} to i_{max}, and the number of different values is a reasonable fraction of i_{max} - i_{min}, then we can construct an array of labels, with the label of the statement for value j in the entry of the table with offset j - i_{min} and the label for the default in entries not filled otherwise. To perform switch,

evaluate the expression to obtain the value of j , check the value is within range and transfer to the table entry at offset $j - i_{min}$.

Syntax-Directed Translation of Case Statements:

Consider the following switch statement:

switch *E*
    begin
        case V_1 : S_1
        case V_2 : S_2
            . . .
        case V_{n-1} : S_{n-1}
        default : S_n
    end

This case statement is translated into intermediate code that has the following form :

Translation of a case statement

```
            code to evaluate E into t
            goto test
L1 :        code for S1
            goto next
L2 :        code for S2
            goto next
              . . .
Ln-1 :      code for Sn-1
            goto next
Ln :        code for Sn
            goto next
test :      if  t = V1 goto L1
            if  t = V2 goto L2
              . . .
            if  t = Vn-1  goto Ln-1
            goto Ln
next :
```

To translate into above form :

- When keyword **switch** is seen, two new labels **test** and **next,** and a new temporary **t** are generated.
- As expression *E* is parsed, the code to evaluate *E* into **t** is generated. After processing *E* , the jump **goto test** is generated.
- As each **case** keyword occurs, a new label L_i is created and entered into the symbol table. A pointer to this symbol-table entry and the value V_i of case constant are placed on a stack (used only to store cases).

- Each statement **case** $V_i : S_i$ is processed by emitting the newly created label L_i, followed by the code for S_i , followed by the jump **goto next**.
- Then when the keyword **end** terminating the body of the switch is found, the code can be generated for the n-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three-address statements of the form

case V_1 L_1
case V_2 L_2
. . .
case V_{n-1} L_{n-1}
case t L_n
label next

where t is the name holding the value of the selector expression E, and L_n is the label for the default statement.

BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels ***backpatching***.

To manipulate lists of labels, we use three functions :

1. *makelist*(i) creates a new list containing only i, an index into the array of quadruples; *makelist* returns a pointer to the list it has made.
2. *merge*(p_1,p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. *backpatch*(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) | E_1 **and** $M\ E_2$
(3) | **not** E_1
(4) | (E_1)
(5) | **id$_1$ relop id$_2$**
(6) | **true**
(7) | **false**
(8) $M \rightarrow \varepsilon$

Synthesized attributes *truelist* and *falselist* of nonterminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by *E.truelist* and *E.falselist*.

Consider production $E \rightarrow E_1$ **and** $M\ E_2$. If E_1 is false, then E is also false, so the statements on E_1*.falselist* become part of *E.falselist*. If E_1 is true, then we must next test E_2, so the target for the statements E_1*.truelist* must be the beginning of the code generated for E_2. This target is obtained using marker nonterminal M.

Attribute *M.quad* records the number of the first statement of E_2*.code*. With the production M $\rightarrow$ ε we associate the semantic action

{ *M.quad* : = *nextquad* }

The variable *nextquad* holds the index of the next quadruple to follow. This value will be backpatched onto the E_1*.truelist* when we have seen the remainder of the production $E \rightarrow E_1$ **and** $M\ E_2$. The translation scheme is as follows:

(1) $E \rightarrow E_1$ **or** $M\ E_2$ { *backpatch* (E_1*.falselist*, *M.quad*);
E.truelist : = *merge*(E_1*.truelist*, E_2*.truelist*);
E.falselist : = E_2*.falselist* }

(2) $E \rightarrow E_1$ **and** $M\ E_2$ { *backpatch* (E_1*.truelist*, *M.quad*);
E.truelist : = E_2*.truelist*;
E.falselist : = *merge*(E_1*.falselist*, E_2*.falselist*) }

(3) $E \rightarrow$ **not** E_1 { *E.truelist* : = E_1*.falselist*;
E.falselist : = E_1*.truelist*; }

(4) $E \rightarrow (E_1)$ { *E.truelist* : = E_1*.truelist*;
E.falselist : = E_1*.falselist*; }

(5) $E \rightarrow$ $\mathbf{id_1}$ **relop** $\mathbf{id_2}$ { *E.truelist* : = *makelist* (*nextquad*);
E.falselist : = *makelist*(*nextquad* + *1*);
emit('if' $\mathbf{id_1}$*.place* **relop***.op* $\mathbf{id_2}$.place '**goto**_')
emit('**goto**_') }

(6) $E \rightarrow$ **true** { *E.truelist* : = *makelist*(*nextquad*);
emit('**goto**_') }

(7) $E \rightarrow$ **false** { *E.falselist* : = *makelist*(*nextquad*);
emit('**goto**_') }

(8) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

Flow-of-Control Statements:

A translation scheme is developed for statements generated by the following grammar :

(1) $S \rightarrow$ **if** E **then** S
(2) | **if** E **then** S **else** S
(3) | **while** E **do** S
(4) | **begin** L **end**
(5) | A
(6) $L \rightarrow L ; S$
(7) | S

Here S denotes a statement, L a statement list, A an assignment statement, and E a boolean expression. We make the tacit assumption that the code that follows a given statement in execution also follows it physically in the quadruple array. Else, an explicit jump must be provided.

Scheme to implement the Translation:

The nonterminal E has two attributes *E.truelist* and *E.falselist*. *L* and *S* also need a list of unfilled quadruples that must eventually be completed by backpatching. These lists are pointed to by the attributes *L..nextlist* and *S.nextlist*. *S.nextlist* is a pointer to a list of all conditional and unconditional jumps to the quadruple following the statement S in execution order, and *L.nextlist* is defined similarly.

The semantic rules for the revised grammar are as follows:

(1) $S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ *backpatch* (*E.truelist*, M_1*.quad*);
backpatch (*E.falselist*, M_2*.quad*);
S.nextlist : = *merge* (S_1*.nextlist*, *merge* (*N.nextlist*, S_2*.nextlist*)) }

We backpatch the jumps when E is true to the quadruple M_1*.quad*, which is the beginning of the code for S_1. Similarly, we backpatch jumps when E is false to go to the beginning of the code for S_2. The list *S.nextlist* includes all jumps out of S_1 and S_2, as well as the jump generated by N.

(2) $N \rightarrow \varepsilon$ { *N.nextlist* : = *makelist*(*nextquad*);
emit('**goto** _') }

(3) $M \rightarrow \varepsilon$ { *M.quad* : = *nextquad* }

(4) $S \rightarrow$ **if** E **then** $M\ S_1$ { *backpatch*(*E.truelist*, *M.quad*);
S.nextlist : = *merge*(*E.falselist*, S_1*.nextlist*) }

(5) $S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$ { *backpatch*(S_1*.nextlist*, M_1*.quad*);
backpatch(*E.truelist*, M_2*.quad*);
S.nextlist : = *E.falselist*
emit('**goto**' M_1*.quad*) }

(6) $S \rightarrow$ **begin** L **end** { *S.nextlist* : = *L.nextlist* }

(7) $S \rightarrow A$ { *S.nextlist* : = **nil** }

The assignment *S.nextlist* : = **nil** initializes *S.nextlist* to an empty list.

(8) $L \rightarrow L1$; *M S* { *backpatch*(L_1*.nextlist*, *M.quad*);
L.nextlist : = *S.nextlist* }

The statement following L_1 in order of execution is the beginning of *S*. Thus the *L1.nextlist* list is backpatched to the beginning of the code for *S*, which is given by *M.quad*.

(9) $L \rightarrow S$ { *L.nextlist* : = *S.nextlist* }

PROCEDURE CALLS

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement

(1) $S \rightarrow$ **call id** (*Elist*)
(2) *Elist* $\rightarrow$ *Elist* , *E*
(3) *Elist* $\rightarrow$ *E*

Calling Sequences:

The translation for a call includes a calling sequence, a sequence of actions taken on entry to and exit from each procedure. The falling are the actions that take place in a calling sequence :

- When a procedure call occurs, space must be allocated for the activation record of the called procedure.
- The arguments of the called procedure must be evaluated and made available to the called procedure in a known place.
- Environment pointers must be established to enable the called procedure to access data in enclosing blocks.
- The state of the calling procedure must be saved so it can resume execution after the call.
- Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished.
- Finally a jump to the beginning of the code for the called procedure must be generated.

For example, consider the following syntax-directed translation

(1) $S \rightarrow$ **call id** (*Elist*)
{ **for** each item *p* on *queue* **do**
emit (‘ **param**’ *p*);

emit ('call' **id.***place*) }

(2) *Elist* → *Elist* , *E*

{ append *E.place* to the end of *queue* }

(3) *Elist* → *E*

{ initialize *queue* to contain only *E.place* }

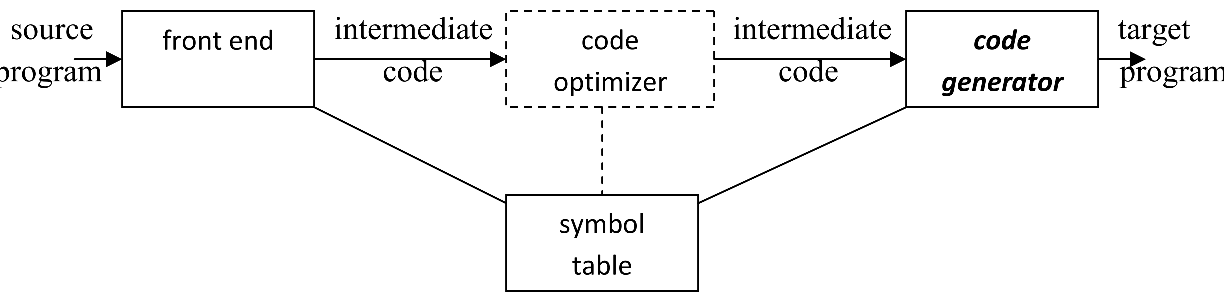
- Here, the code for S is the code for *Elist*, which evaluates the arguments, followed by a **param** *p* statement for each argument, followed by a **call** statement.

- *queue* is emptied and then gets a single pointer to the symbol table location for the name that denotes the value of E.

MODULE-4 CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

- Intermediate representation can be :
  a. Linear representation such as postfix notation
  b. Three address representation such as quadruples
  c. Virtual machine representation such as stack machine code
  d. Graphical representations such as syntax trees and dags.

- Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
    - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language
   - It allows subprograms to be compiled separately.

c. Assembly language
   - Code generation is made easier.

3. Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

- Labels in three-address statements have to be converted to addresses of instructions. For example,
  j : **goto** i generates jump instruction as follows :
  - if $i < j$, a backward jump instruction with target address equal to location of code for quadruple i is generated.
  - if $i > j$, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

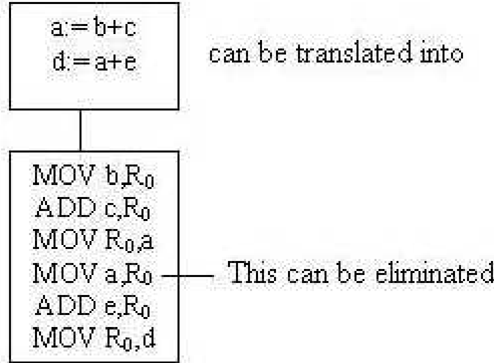
4. Instruction selection:

- The instructions of target machine should be complete and uniform.

- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

- The quality of the generated code is determined by its speed and size.

- The former statement can be translated into the latter statement as shown below:



5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.

- The use of registers is subdivided into two subproblems :
  - ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

- ➢ ***Register assignment*** – the specific register that a variable will reside in is picked.

- Certain machine requires even-odd *register pairs* for some operands and results.
  For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register pair
  y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, $R_0, R_1, \ldots, R_{n-1}$.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :
  MOV (move *source* to *destination*)
  ADD (add *source* to *destination*)
  SUB (subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	FORM	*ADDRESS*	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ;
  MOV $4(R_0)$, M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

  i) MOV b, R_0
  ADD c, R_0 cost = 6
  MOV R_0, a

  ii) MOV b, a
  ADD c, a cost = 6

  iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :
  MOV $*R_1$, $*R_0$
  ADD $*R_2$, $*R_0$ cost = 2

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

RUN-TIME STORAGE MANAGEMENT

- Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.
- The two standard storage allocation strategies are:
  1. Static allocation
  2. Stack allocation
- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.
- The following three-address statements are associated with the run-time allocation and deallocation of activation records:
  1. Call,
  2. Return,
  3. Halt, and
  4. Action, a placeholder for other statements.
- We assume that the run-time memory is divided into areas for:
  1. Code
  2. Static data
  3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* + 20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,
callee.static_area – Address of the activation record
callee.code_area – Address of the first instruction for called procedure
#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :

GOTO **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV *#stackstart* , SP /* initializes stack */

Code for the first procedure

HALT /* terminate execution */

Implementation of Call statement:

ADD *#caller.recordsize*, SP /* increment stack pointer */

MOV *#here* + 16, *SP /*Save return address */

GOTO *callee.code_area*

where,
caller.recordsize – size of the activation record
#here + 16 – address of the instruction following the **GOTO**

Implementation of Return statement:

GOTO *0 (SP) /*return to the caller */

SUB *#caller.recordsize*, SP /* decrement SP and restore to previous value */

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
  t_1 : = a * a
  t_2 : = a * b
  t_3 : = 2 * t_2
  t_4 : = t_1 + t_3
  t_5 : = b * b
  t_6 : = t_4 + t_5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin

        prod :=0;

        i:=1;

        do begin

                prod :=prod+ a[i] * b[i];

                i :=i+1;

        end

        while i <= 20

end
```

- The three-address code for the above source program is given as :

```
(1)     prod := 0

(2)     i := 1

(3)     t1 := 4* i

(4)     t2 := a[t1]          /*compute a[i] */

(5)     t3 := 4* i

(6)     t4 :=  b[t3]         /*compute b[i] */

(7)     t5 := t2*t4

(8)     t6 := prod+t5

(9)     prod := t6

(10)    t7 := i+1

(11)    i := t7

(12)    if i<=20 goto (3)
```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:

a : = b + c b : = a – d c : = b + c d : = a – d	⟶	a : = b + c b : = a - d c : = b + c d : = b

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) Dead-code elimination:

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables:

A statement **t : = b + c** (t is a temporary) can be changed to **u : = b + c** (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block.
Such a block is called a *normal-form block*.

d) Interchange of statements:

Suppose a block has the following two adjacent statements:

t1 : = b + c
t2 : = x + y

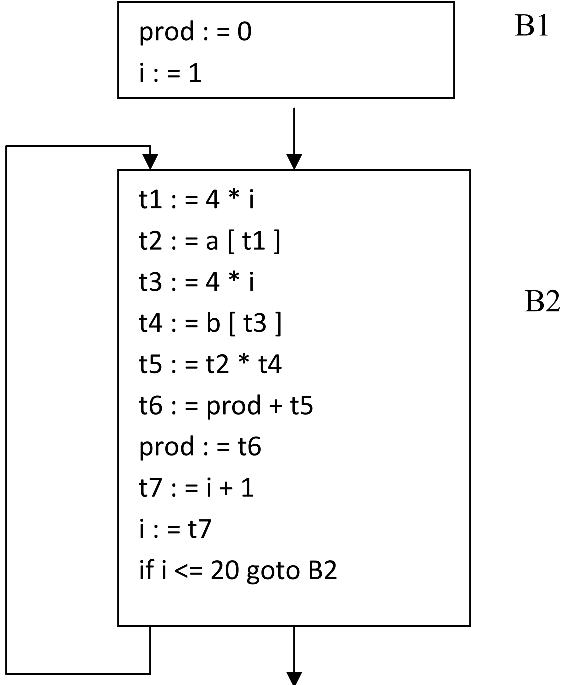
We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is $\mathbf{t_1}$ and neither **b** nor **c** is $\mathbf{t_2}$.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
Examples:
i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



- B_1 is the *initial* node. B_2 immediately follows B_1, so there is an edge from B_1 to B_2. The target of jump from last statement of B_1 is the first statement B_2, so there is an edge from B_1 (last statement) to B_2 (first statement).
- B_1 is the *predecessor* of B_2, and B_2 is a *successor* of B_1.

Loops

- A loop is a collection of nodes in a flow graph such that
    1. All nodes in the collection are *strongly connected*.
    2. The collection of nodes has a unique *entry*.
- A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Input: Basic block B of three-address statements

Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

Symbol Table:

Names	Liveliness	Next-use
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
  It can have the following sequence of codes:

ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

(or)

ADD c, R_i Cost = 2 // if c is in a memory location

(or)

MOV c, R_j Cost = 3 // move c from memory to Rj and add

ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

  t : = a – b
  u : = a – c
  v : = t + u
  d : = v + u

with d live at the end.

Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0

d : = v + u	ADD R_1, R_0
	
	MOV R_0, d
		R_0 contains d	d in R_0
			d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	**Code Generated**	**Cost**
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	**Code Generated**	**Cost**
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	**Code**
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */

x : = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0,x
	CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
    1. Leaves are labeled by unique identifiers, either variable names or constants.
    2. Interior nodes are labeled by an operator symbol.
    3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

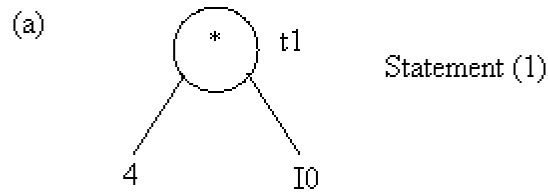
identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

1. t_1 := 4* i
2. t_2 := a[t_1]
3. t_3 := 4* i
4. t_4 := b[t_3]
5. t_5 := t_2*t_4
6. t_6 := prod+t_5
7. prod := t_6
8. t_7 := i+1
9. i := t_7
10. if i<=20 goto (1)

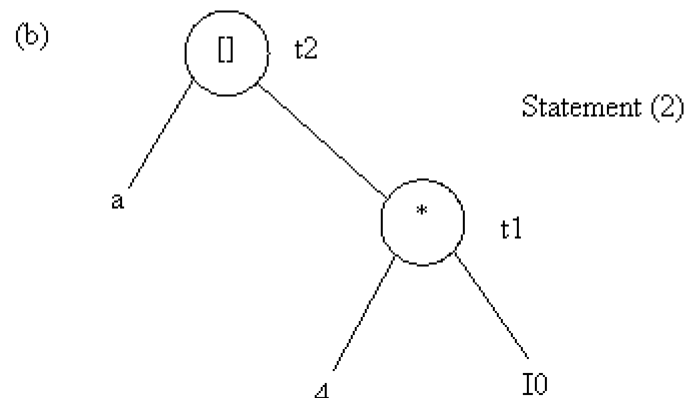
Stages in DAG Construction

(a)



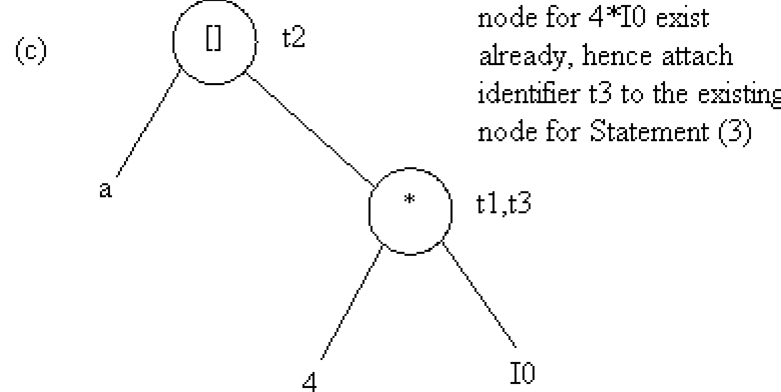
Statement (1)

(b)



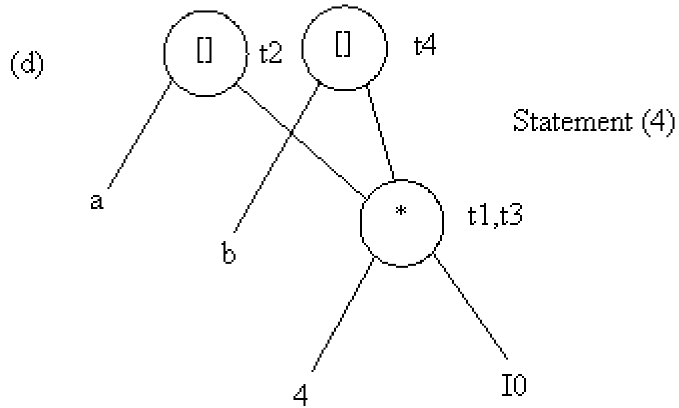
Statement (2)

(c)

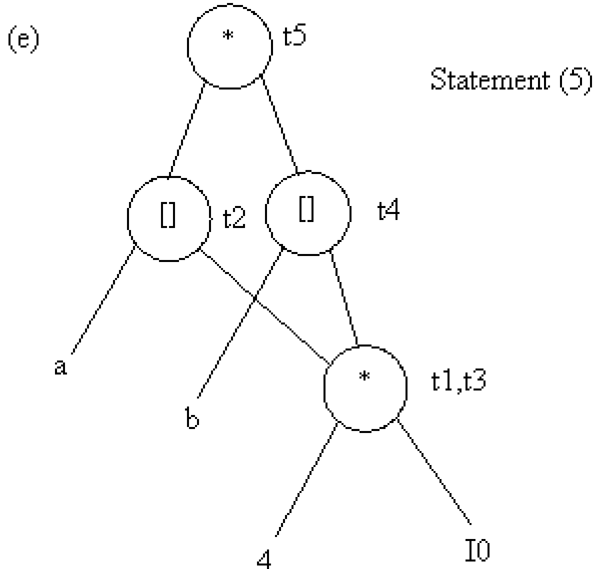


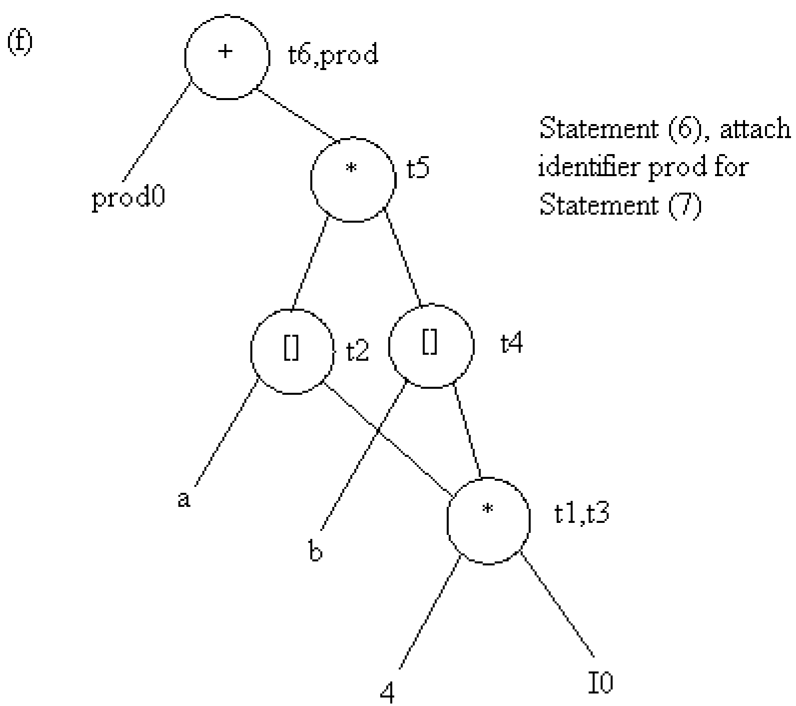
node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

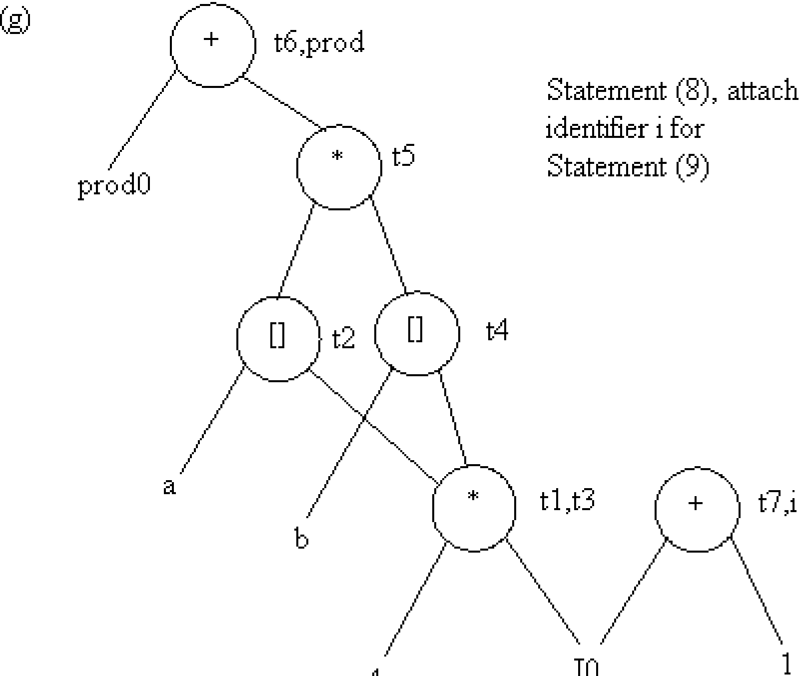
(d)

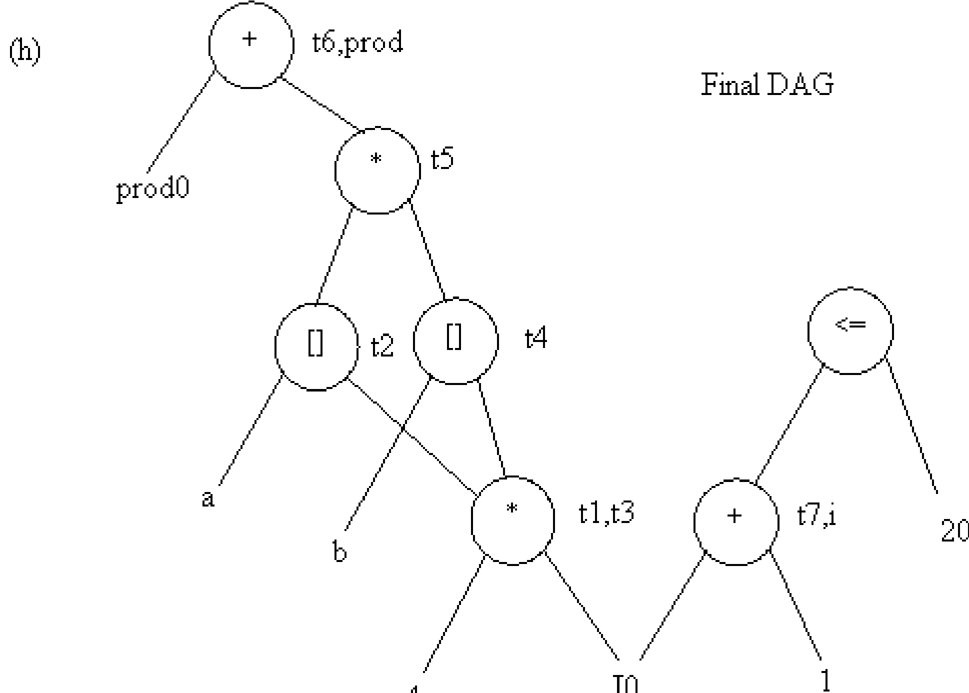


Statement (4)


(e)
*
t5
Statement (5)
[]
t2
[]
t4
a
b
*
t1,t3
4
I0


(f)
+
t6,prod
Statement (6), attach identifier prod for Statement (7)
prod0
*
t5
[]
t2
[]
t4
a
b
*
t1,t3
4
I0





Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

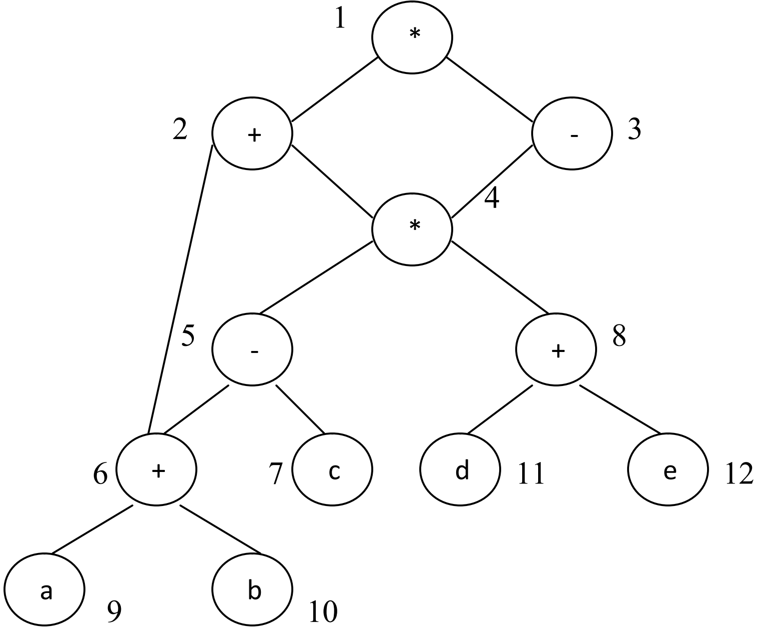
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

```
1)  while unlisted interior nodes remain do begin
2)       select an unlisted node n, all of whose parents have been listed;
3)       list n;
4)       while the leftmost child m of n has no unlisted parents and is not a leaf do
             begin
5)               list m;
6)               n : = m
             end
         end
```

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t_8 : = d + e
t_6 : = a + b
t_5 : = t_6 – c
t_4 : = t_5 * t_8
t_3 : = t4 – e
t_2 : = t_6 + t_4
t_1 : = t_2 * t_3

This will yield an optimal code for the DAG on machine whatever be the number of registers.

MODULE-4 - CODE OPTIMIZATION

INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

- Optimizations are classified into two categories. They are
  - Machine independent optimizations:
  - Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

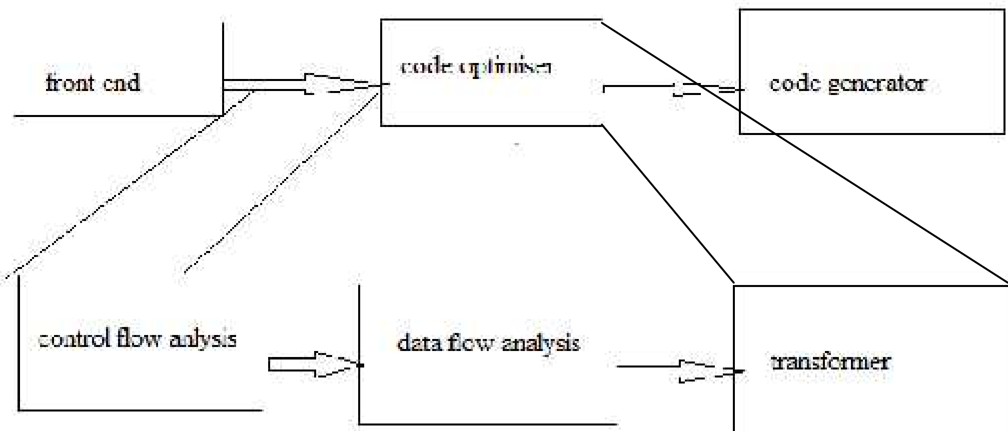
- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
  - control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations

  ✓ Common sub expression elimination,
  ✓ Copy propagation,
  ✓ Dead-code elimination, and
  ✓ Constant folding

  are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

- **Common Sub expressions elimination:**

  - An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
  - For example

t_1: = 4*i
t_2: = a [t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

- **Copy Propagation:**

  - Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.
  - For example:

x=Pi;
……
A=x*r*r;

The optimization using copy propagation can be done as follows:

A=Pi*r*r;

Here the variable x is eliminated

- **Dead-Code Eliminations:**

  - A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.
Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, ‘if’ statement is dead code because this condition will never get satisfied.

- **Constant folding**:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

- One advantage of copy propagation is that it often turns the copy statement into dead code.
- ✓ For example,
  a=3.14157/2 can be replaced by
  a=1.570 there by eliminating a division operation.

- **Loop Optimizations:**

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:

- ✓ code motion, which moves code outside a loop;
- ✓ Induction-variable elimination, which we apply to replace variables from inner loop.
- ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

- **Code Motion:**

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2)     /* statement does not change limit*/
```

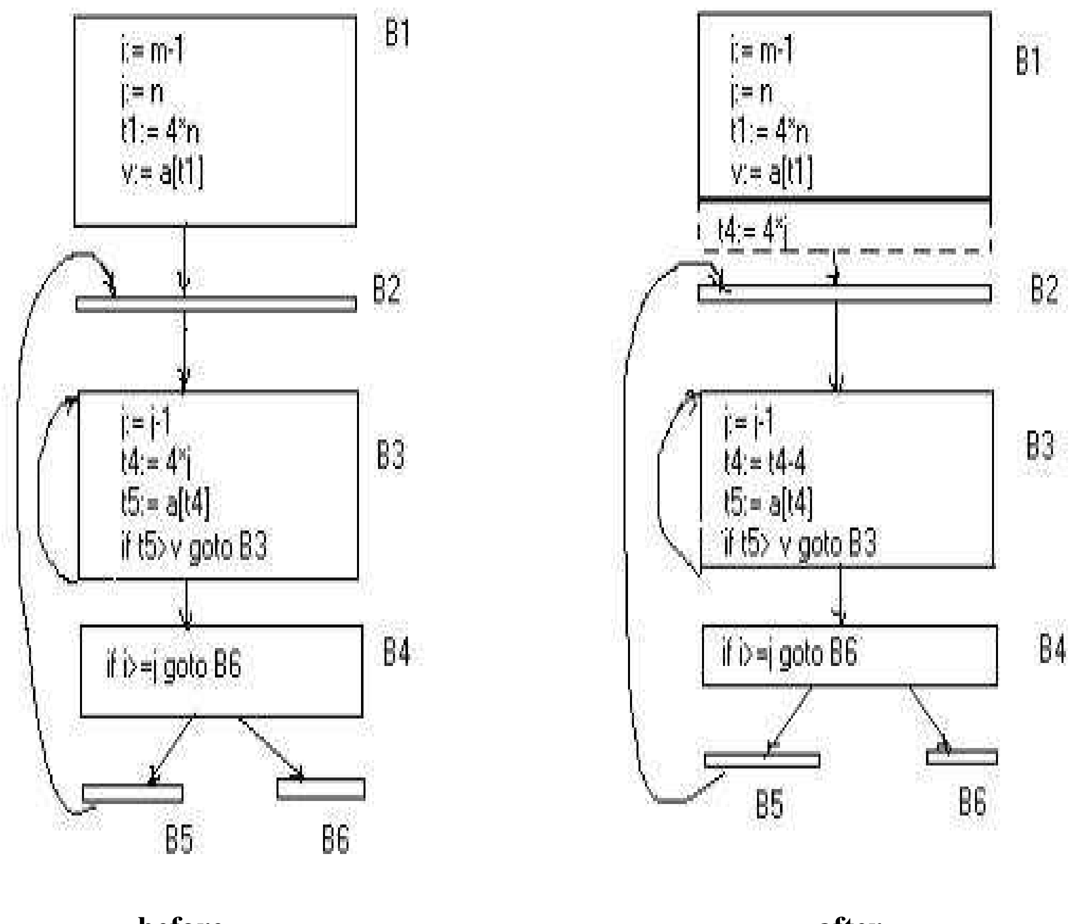
Code motion will result in the equivalent of

t= limit-2;
while (i<=t)   /* statement does not change limit or t */

➢ **Induction Variables :**

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:
As the relationship t_4:=4*j surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is



before **after**

initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

- **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- ✓ Structure-Preserving Transformations
- ✓ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

- **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered aga in – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

- **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

- **Renaming of temporary variables:**

  - A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.
  - In this we can transform a basic block to its equivalent block called normal-form block.

- **Interchange of two independent adjacent statements:**

  - Two statements

    t_1:=b+c

    t_2:=x+y

    can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

  a :=b+c
  e :=c+d+b

  the following intermediate code may be generated:

  a :=b+c
  t :=c+d
  e :=t+b

- Example:

  x:=x+0 can be removed

  x:=y**2 can be replaced by a cheaper statement x:=y*y

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

LOOPS IN FLOW GRAPH

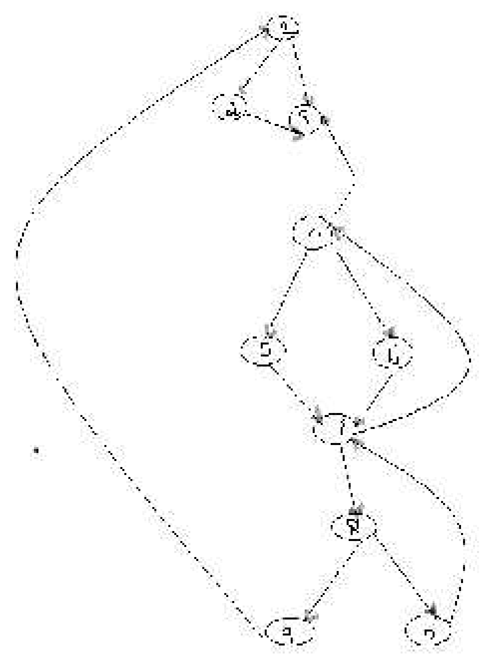
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

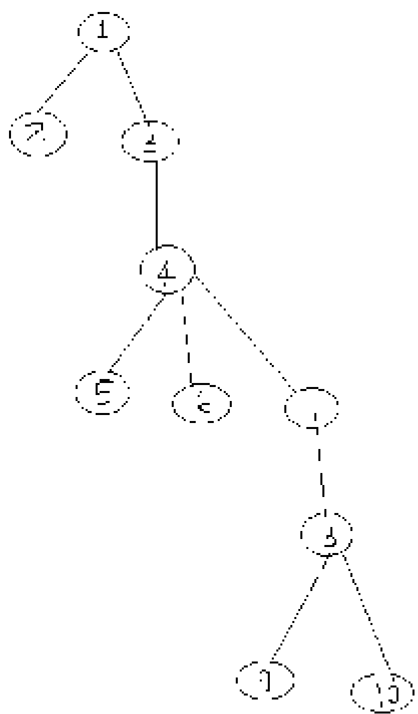
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*In the flow graph below,
*Initial node,node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are

  - ✓ A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - ✓ There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

  - ✓ Example:

    In the above graph,

    7 → 4 4 DOM 7

    10 →7 7 DOM 10

    4 → 3

    8 → 3

    9 →1

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus th e set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedure insert(m);
if m is not in *loop* **then begin**
    loop := *loop* U {m};
    push *m* onto *stack*
end;

stack : = empty;

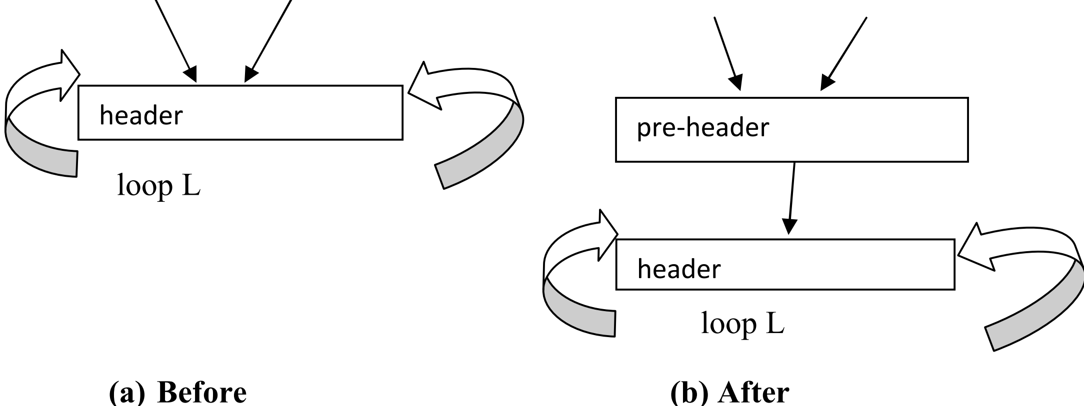
loop : = {*d*};
insert(*n*);
while *stack* is not empty **do begin**
    pop *m*, the first element of *stack*, off *stack*;
    for each predecessor *p* of *m* **do** *insert(p)*
end

Inner loop:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:

  A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

✓ The forward edges from an acyclic graph in which every node can be reached from initial node of G.

✓ The back edges consist only of edges where heads dominate theirs tails.

✓ Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- The remaining edges are forward edges.

- If the forward edges form an acyclic graph, then we can say the flow graph reducible.

- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.

- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  ✓ Redundant-instructions elimination
  ✓ Flow-of-control optimizations
  ✓ Algebraic simplifications
  ✓ Use of machine idioms
  ✓ Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug 0
….
If ( debug ) {
Print debugging information
}
```

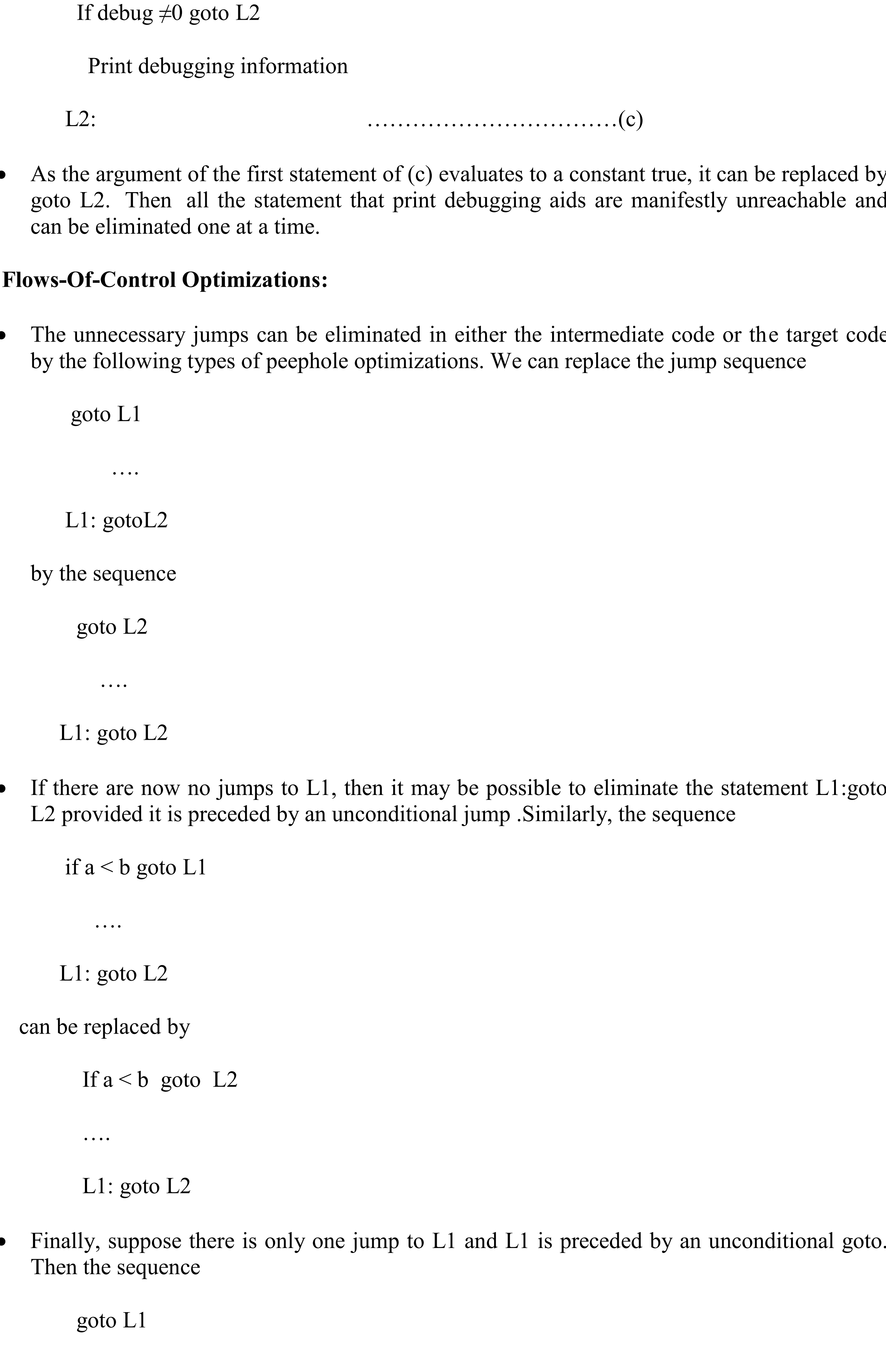
- In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
  goto L2
L1: print debugging information
L2:                              …………………………(a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2:                              ……………………………(b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by



If debug ≠0 goto L2

Print debugging information

L2: ……………………………(c)

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

….

L1: gotoL2

by the sequence

goto L2

….

L1: goto L2

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1: goto L2

can be replaced by

If a < b goto L2

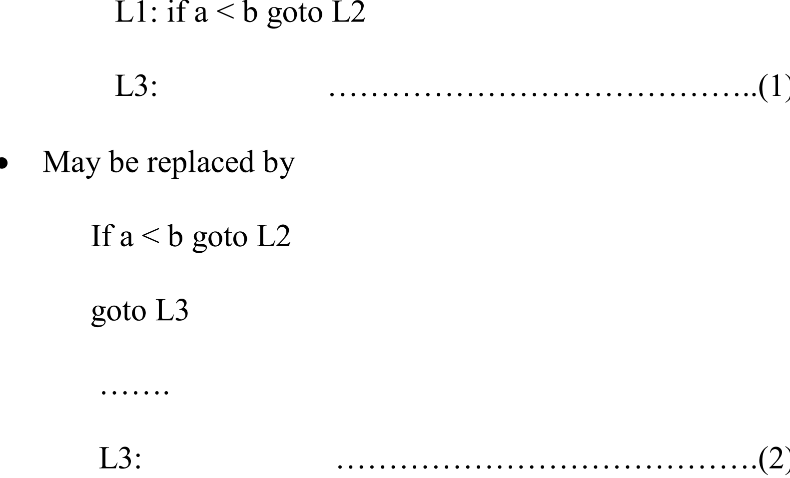
….

L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

……..



```
L1: if a < b goto L2

L3:                 …………………………………..(1)
```

- May be replaced by

```
If a < b goto L2

goto L3

…….

L3:                 ………………………………….(2)
```

- While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

Algebraic Simplification:

- There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

```
x := x+0

Or

x := x * 1
```

- Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$$X^2 \rightarrow X*X$$

Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++

i:=i-1 → i- -

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

- The details of how data-flow equations are set and solved depend on three factors.

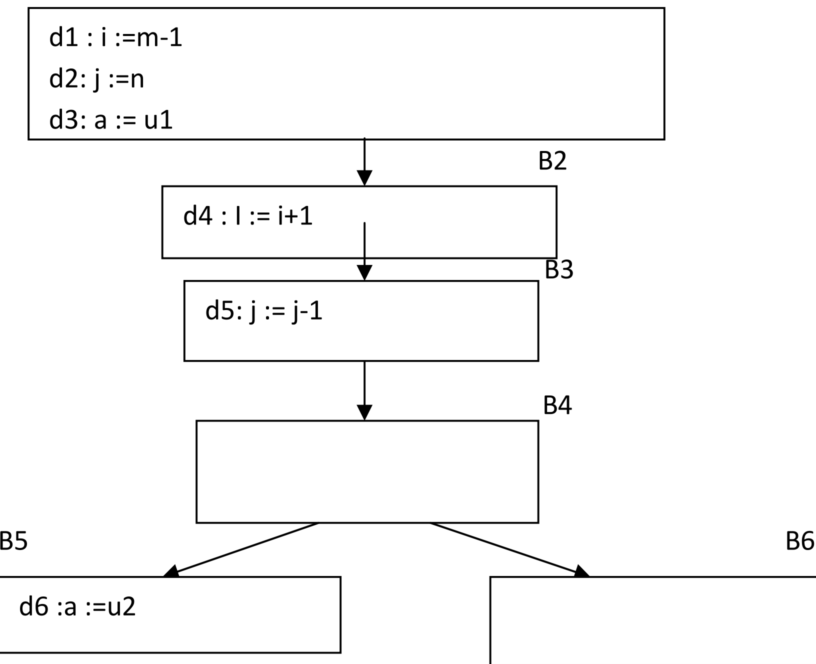
✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points p_1, p_2,….,p_n such that for each i between 1 and n-1, either

✓ P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.

- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.

✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached

by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

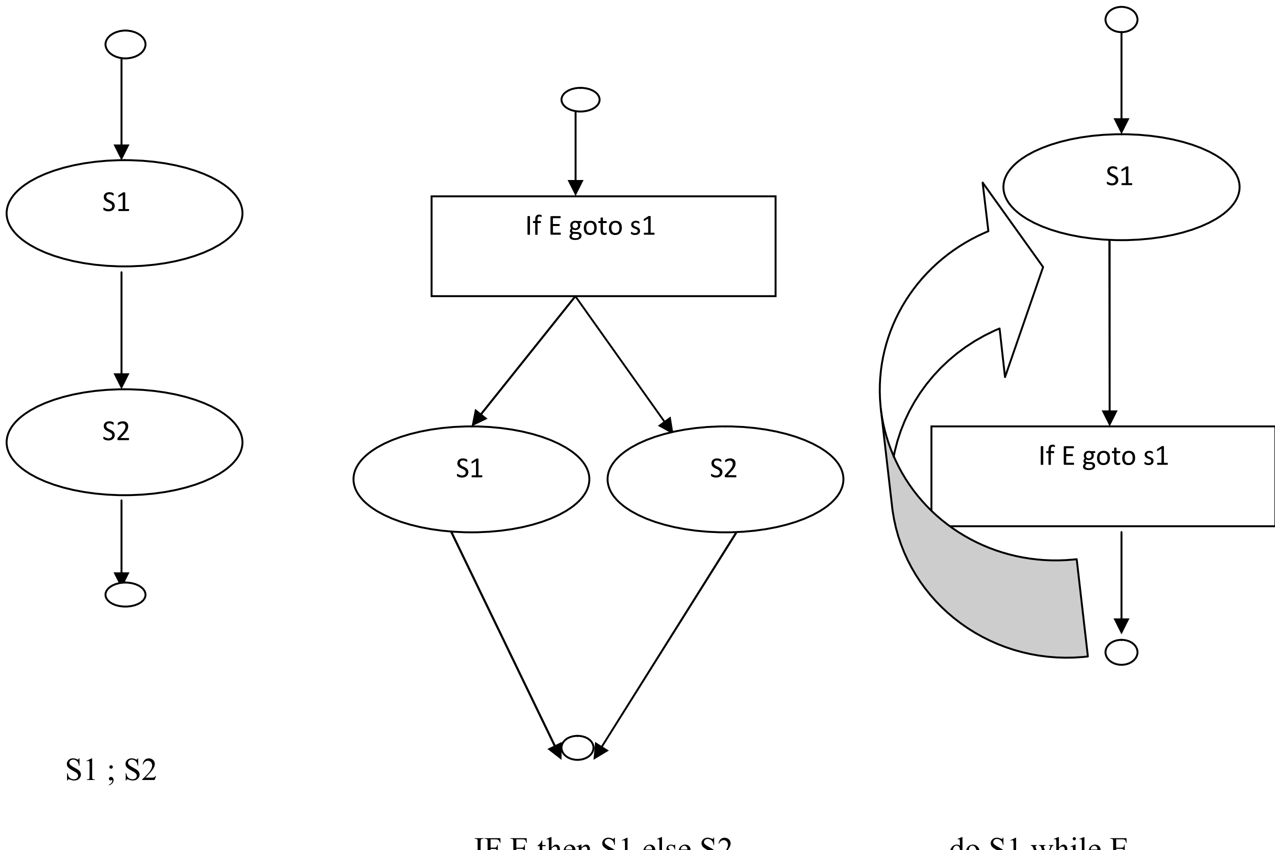
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  S ⟶ id: = E| S; S | if E then S else S | do S while E

  E ⟶ id + id| id

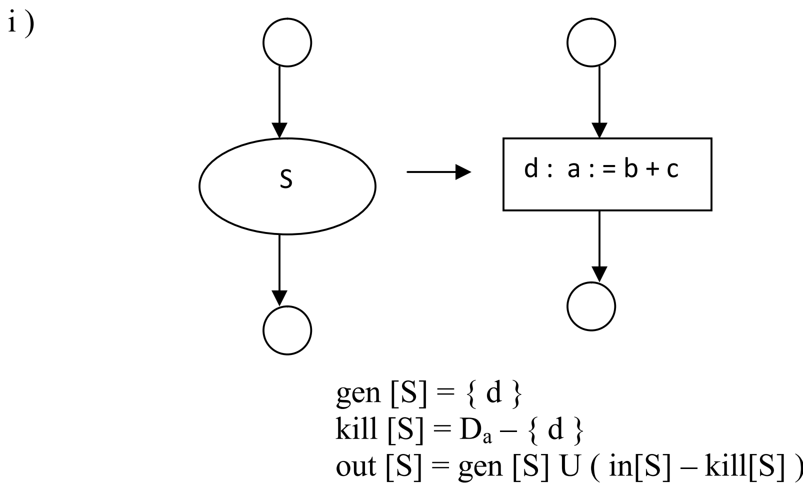
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus

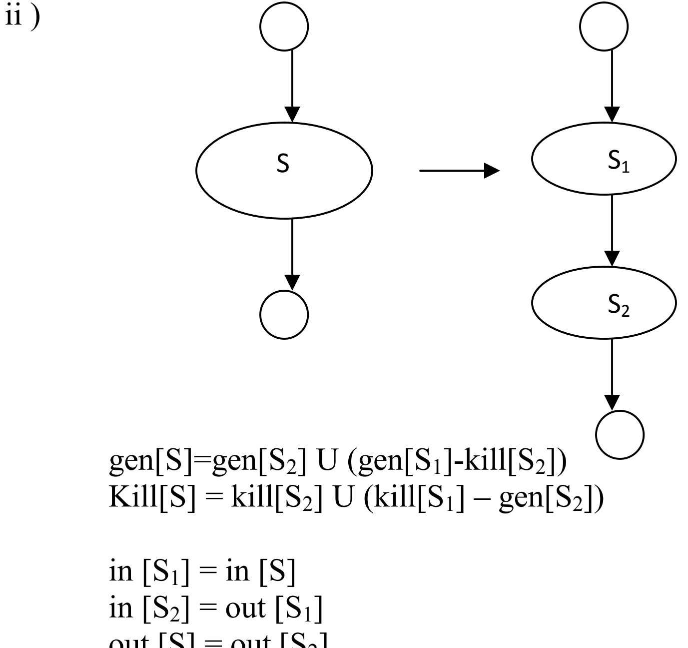
Gen[S]={d}

- On the other hand, d "kills" all other definitions of a, so we write

Kill[S] = D_a – {d}

Where, D_a is the set of all definitions in the program for variable a.

ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write

  gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])

- Similar reasoning applies to the killing of a definition, so we have

  Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.

- Assuming we know in[S] we compute out by equation, that is

  Out[S] = gen[S] U (in[S] - kill[S])

- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].

- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

  In[S_1] = in[S_2] = in[S]

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

  Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming

languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

CODE IMPROVIG TRANSFORMATIONS

- Algorithms for performing the code improving transformations rely on data-flow information. Here we consider common sub-expression elimination, copy propagation and transformations for moving loop invariant computations out of loops and for eliminating induction variables.

- Global transformations are not substitute for local transformations; both must be performed.

Elimination of global common sub expressions:

- The available expressions data-flow problem discussed in the last section allows us to determine if an expression at point p in a flow graph is a common sub-expression. The following algorithm formalizes the intuitive ideas presented for eliminating common sub-expressions.

❖ **ALGORITHM: Global common sub expression elimination.**

INPUT: A flow graph with available expression information.

OUTPUT: A revised flow graph.

METHOD: For every statement s of the form x := y+z[6] such that y+z is available at the beginning of block and neither y nor r z is defined prior to statement s in that block, do the following.

✓ **To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.**

✓ **Create new variable u.**

✓ **Replace each statement w: =y+z found in (1) by**
u : = y + z
w : = u

✓ **Replace statement s by x:=u.**

- Some remarks about this algorithm are in order.

✓ The search in step(1) of the algorithm for the evaluations of y+z that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.

- ✓ Not all changes made by algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (1), probably to one.
- ✓ Algorithm will miss the fact that a*z and c*z must have the same value in

```
a :=x+y                c :=x+y

           vs

b :=a*z                d :=c*z
```

- ✓ Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

Copy propagation:

- Various algorithms introduce copy statements such as x :=copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x.
- Statement s must be the only definition of x reaching u.
- On every path from s to including paths that go through u several times, there are no assignments to y.
- Condition (1) can be checked using ud-changing information. We shall set up a new data-flow analysis problem in which in[B] is the set of copies s: x:=y such that every path from initial node to the beginning of B contains the statement s, and subsequent to the last occurrence of s, there are no assignments to y.

❖ **ALGORITHM: Copy propagation.**

INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, and with c_in[B] representing the solution to equations that is the set of copies x:=y that reach block B along every path, with no assignment to x or y following the last occurrence of x:=y on the path. We also need ud-chains giving the uses of each definition.

OUTPUT: A revised flow graph.

METHOD: For each copy s : x:=y do the following:

- ✓ **Determine those uses of x that are reached by this definition of namely, s: x: =y.**
- ✓ **Determine whether for every use of x found in (1) , s is in c_in[B], where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B]then s is the only definition of x that reaches B.**

- ✓ **If s meets the conditions of (2), then remove s and replace all uses of x found in (1) by y.**

Detection of loop-invariant computations:

- Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.
- If an assignment x := y+z is at a position in the loop where all possible definitions of y and z are outside the loop, then y+z is loop-invariant because its value will be the same each time x:=y+z is encountered. Having recognized that value of x will not change, consider v := x+w, where w could only have been defined outside the loop, then x+w is also loop-invariant.

❖ **ALGORITHM: Detection of loop-invariant computations.**

INPUT: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.

OUTPUT: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.

METHOD: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

- ✓ **Mark "invariant" those statements whose operands are all either constant or have all their reaching definitions outside L.**
- ✓ **Repeat step (3) until at some repetition no new statements are marked "invariant".**
- ✓ **Mark "invariant" all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.**

Performing code motion:

- Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider s: x: =y+z.

- ✓ The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
- ✓ There is no other statement in the loop that assigns to x. Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.

- ✓ No use of x in the loop is reached by any definition of x other than s. This condition too will be satisfied, normally, if x is temporary.

❖ **ALGORITHM: Code motion.**

INPUT: A loop L with ud-chaining information and dominator information.

OUTPUT: A revised version of the loop with a pre-header and some statements moved to the pre-header.

METHOD:

- ✓ **Use loop-invariant computation algorithm to find loop-invariant statements.**
- ✓ **For each statement s defining x found in step(1), check:**
  - **i) That it is in a block that dominates all exits of L,**
  - **ii) That x is not defined elsewhere in L, and**
  - **iii) That all uses in L of x can only be reached by the definition of x in statement s.**
- ✓ **Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii) , to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.**

- To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L. When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L. Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s.

Alternative code motion strategies:

- The condition (1) can be relaxed if we are willing to take the risk that we may actually increase the running time of the program a bit; of course, we never change what the program computes. The relaxed version of code motion condition (1) is that we may move a statement s assigning x only if:

  1’. The block containing s either dominates all exists of the loop, or x is not used outside the loop. For example, if x is a temporary variable, we can be sure that the value will be used only in its own block.

- If code motion algorithm is modified to use condition (1’), occasionally the running time will increase, but we can expect to do reasonably well on the average. The modified algorithm may move to pre-header certain computations that may not be executed in the

loop. Not only does this risk slowing down the program significantly, it may also cause an error in certain circumstances.

- Even if none of the conditions of (2i), (2ii), (2iii) of code motion algorithm are met by an assignment x: =y+z, we can still take the computation y+z outside a loop. Create a new temporary t, and set t: =y+z in the pre-header. Then replace x: =y+z by x: =t in the loop. In many cases we can propagate out the copy statement x: = t.

Maintaining data-flow information after code motion:

- The transformations of code motion algorithm do not change ud-chaining information, since by condition (2i), (2ii), and (2iii), all uses of the variable assigned by a moved statement s that were reached by s are still reached by s from its new position.
- Definitions of variables used by s are either outside L, in which case they reach the pre-header, or they are inside L, in which case by step (3) they were moved to pre-header ahead of s.
- If the ud-chains are represented by lists of pointers to pointers to statements, we can maintain ud-chains when we move statement s by simply changing the pointer to s when we move it. That is, we create for each statement s pointer $p_{s,}$ which always points to s.
- We put the pointer on each ud-chain containing s. Then, no matter where we move s, we have only to change p_s, regardless of how many ud-chains s is on.
- The dominator information is changed slightly by code motion. The pre-header is now the immediate dominator of the header, and the immediate dominator of the pre-header is the node that formerly was the immediate dominator of the header. That is, the pre-header is inserted into the dominator tree as the parent of the header.

Elimination of induction variable:

- A variable x is called an induction variable of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as in a loop headed by for i := 1 to 10.
- However, our methods deal with variables that are incremented or decremented zero, one, two, or more times as we go around a loop. The number of changes to an induction variable may even differ at different iterations.
- A common situation is one in which an induction variable, say i, indexes an array, and some other induction variable, say t, whose value is a linear function of i, is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t.
- We shall look for basic induction variables, which are those variables i whose only assignments within loop L are of the form i := i+c or i-c, where c is a constant.

❖ **ALGORITHM: Elimination of induction variables.**

INPUT: A loop L with reaching definition information, loop-invariant computation information and live variable information.

OUTPUT: A revised loop.

METHOD:

- ✓ **Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i's family, preferably one such that c and d in its triple are as simple as possible and modify each test that i appears in to use j instead. We assume in the following tat c is positive. A test of the form 'if i relop x goto B', where x is not an induction variable, is replaced by**

  r := c*x   /* r := x if c is 1. */

  r := r+d   /* omit if d is 0 */

  if j relop r goto B

  where, r is a new temporary. The case 'if x relop i goto B' is handled analogously. If there are two induction variables i_1 and i_2 in the test if i_1 relop i_2 goto B, then we check if both i_1 and i_2 can be replaced. The easy case is when we have j_1 with triple and j_2 with triple, and $c_1=c_2$ and $d_1=d_2$. Then, i_1 relop i_2 is equivalent to j_1 relop j_2.

- ✓ **Now, consider each induction variable j for which a statement j: =s was introduced. First check that there can be no assignment to s between the introduced statement j :=s and any use of j. In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the check. Then replace all uses of j by uses of s and delete statement j: =s.**